DTIC FILE COPY

(2)

AD-A223 692

:NTATION PAGE

Form Approved
OPM No. 0704-0188

...verage 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and ...Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions ...te for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to ...f Budget, Washington, DC 20503.

...ORT DATE	3. REPORT TYPE AND DATES COVERED
	Final 31 May 89 to 31 May 90

4. TITLE AND SUBTITLE Ada Compiler Validation Summary Report: Alsys Limited, AlsyCOMP_017 V4.0, MicroVAX II (Host) to INMOS T800 implemented on a B403 TRAM (Target), 890531N1.10088

5. FUNDING NUMBERS

6. AUTHOR(S)
National Computing Centre Limited
Manchester, UNITED KINGDOM

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
National Computing Centre Limited
Oxford Road
Manchester MI 7ED
UNITED KINGDOM

8. PERFORMING ORGANIZATION REPORT NUMBER
AVF-VSR-90502/50

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Ada Joint Program Office
United States Department of Defense
Washington, D.C. 20301-3081

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

Alsys Limited, AlsyCOMP_017 V4.0, Manchester England, MicroVAX II under MicroVMS V4.7 (Host) to INMOS T800 implemented on a B403 TRAM (bare)(Target), ACVC 1.10.

DTIC ELECTE JUN 27 1990 S B D

14. SUBJECT TERMS Ada programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, Validation Testing, Ada Validation Office, Ada Validation Facility, ANSI/MIL-STD-1815A, Ada Joint Program Office

15. NUMBER OF PAGES

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	

NSN 7540-01-280-5500

Standard Form 298, (Rev. 2-89)
Prescribed by ANSI Std. 239-18
298-01

90 06 25 124

AVF Control Number: AVF-VSR-90502/50

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: #890531N1.10088
Alsys Limited
AlsyCOMP_017 V4.0
MicroVAX II Host and INMOS T800 implemented on a B403 TRAM Target

Completion of On-Site Testing:
31 May 1989

Prepared By:
Testing Services
The National Computing Centre Limited
Oxford Road
Manchester M1 7ED
England

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington DC 20301-3081

Ada Compiler Validation Summary Report:

Compiler Name: AlsyCOMP_017 V4.0

Certificate Number: #890531N1.10088

Host: MicroVAX II under MicroVMS V4.7

Target: INMOS T800 implemented on a B403 TRAM (bare)

Testing Completed 31 May 1989 Using ACVC 1.10

This report has been reviewed and is approved.

J Pink

Jane Pink
Testing Services Manager
The National Computing Centre Limited
Oxford Road
Manchester M1 7ED
England

Ada Validation Organization
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria VA 22311

John F. Solomond

Ada Joint Program Office
Dr John Solomond
Director AJPO
Department of Defense
Washington DC 20301

TABLE OF CONTENTS

CHAPTER 1
INTRODUCTION 1
1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT 1
1.2 USE OF THIS VALIDATION SUMMARY REPORT 2
1.3 REFERENCES 2
1.4 DEFINITION OF TERMS 3
1.5 ACVC TEST CLASSES 4

CHAPTER 2
CONFIGURATION INFORMATION 1
2.1 CONFIGURATION TESTED 1
2.2 IMPLEMENTATION CHARACTERISTICS 2

CHAPTER 3
TEST INFORMATION 1
3.1 TEST RESULTS 1
3.2 SUMMARY OF TEST RESULTS BY CLASS 1
3.3 SUMMARY OF TEST RESULTS BY CHAPTER 1
3.4 WITHDRAWN TESTS 1
3.5 INAPPLICABLE TESTS 2
3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS . . . 6
3.7 ADDITIONAL TESTING INFORMATION 6
3.7.1 Prevalidation 7
3.7.2 Test Method 7
3.7.3 Test Site 8

APPENDIX A
DECLARATION OF CONFORMANCE 1

APPENDIX B
APPENDIX F OF THE Ada STANDARD 1

APPENDIX C
TEST PARAMETERS 1

APPENDIX D
WITHDRAWN TESTS 1

CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies -- for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation dependent, but is permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

- To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard
- To attempt to identify any language constructs not supported by the compiler but required by the Ada Standard
- To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by The National Computer Centre Limited according to procedures established by the Ada Joint Program Office and administered by the Ada Validation

Organization (AVO). On-site testing was completed **31 May 1989** at **Alsys Limited, Partridge House, Newtown Road, Henley-on-Thames, Oxfordshire, RG9 1EN, UNITED KINGDOM.**

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C. #552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

Ada Information Clearinghouse
Ada Joint Program Office
OUSDRE
The Pentagon, Rm 3D-139 (Fern Stre
Washington DC 20301-3081

or from:

Testing Services
The National Computing Centre Limited
Oxford Road
Manchester M1 7ED
England

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.3 REFERENCES

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.

2. Ada Compiler Validation Procedures and Guidelines, Ada Joint Program Office, 1 January 1987.

3. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., December 1986.

4. Ada Compiler Validation Capability User's Guide, December 1986.

1.4 DEFINITION OF TERMS

ACVC	The Ada Compiler Validation Capability. The set of Ada programs that tests the conformity of an Ada compiler to the Ada programming language.
Ada Commentary	An Ada Commentary contains all information relevant to the point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.
Ada Standard	ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
Applicant	The agency requesting validation.
AVF	The Ada Validation Facility. The AVF is responsible for conducting compiler validations according to procedures contained in the Ada Compiler Validation Procedures and Guidelines.
AVO	The Ada Validation Organization. The AVO has oversight authority over all AVF practices for the purpose of maintaining a uniform process for validation of Ada compilers. The AVO provides administrative and technical support for Ada validations to ensure consistent practices.
Compiler	A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.
Failed test	An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.
Host	The computer on which the compiler resides.
Inapplicable test	An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.
Passed test	An ACVC test for which a compiler generates the expected result.

Target	The computer which executes the code generated by the compiler.
Test	A program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.
Withdrawn test	An ACVC test found to be incorrect and not used to check conformity to the Ada Standard. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce errors because of the way in which a program library is used at link time.

Class A tests ensure the successful compilation and execution of legal Ada programs with certain language constructs which cannot be verified at run time. There are no explicit program components in a Class A test to check semantics. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check the run time system to ensure that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters -- for example, the number of identifiers permitted in a compilation or the number of units in a library -- a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Class E tests are expected to execute successfully and check implementation-dependent options and resolutions of ambiguities in the Ada Standard. Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time -- that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated. In some cases, an implementation may legitimately detect errors during compilation of the test.

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for Chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of each test in the ACVC follows conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and place features that may not be supported by all implementations in separate tests. However, some tests contain values that require the test to be customized according to implementation-specific values -- for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of this validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: **AlsyCOMP_017 V4.0**

ACVC Version: **1.10**

Certificate Number: **#890531N1.10088**

Host Computer:

Machine: **MicroVAX II**

Operating System: **MicroVMS V4.7**

Memory Size: **9Mb**

Target Computer:

Machine: **INMOS T800 implemented on a B403 TRAM (bare), using the Host running INMOS Iserver V1.30 for file-server support.**

Memory Size: **1Mb**

Communications Network: **CAPLIN QT0 Board**

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. The tests demonstrate the following characteristics:

a. Capacities.

(1) The compiler correctly processes a compilation containing 723 variables in the same declarative part. (See test D29002K.)

(2) The compiler correctly processes tests containing loop statements nested to 65 levels. (See tests D55A03A..H (8 tests).)

(3) The compiler correctly processes tests containing block statements nested to 65 levels. (See test D56001B.)

(4) The compiler correctly processes tests containing recursive procedures separately compiled as subunits nested to 17 levels. (See tests D64005E..G (3 tests).)

b. Predefined types.

(1) This implementation supports the additional predefined types SHORT_INTEGER and LONG_FLOAT in the package STANDARD. (See tests B86001T..Z (7 tests).)

c. Based literals.

(1) An implementation is allowed raise NUMERIC_ERROR or CONSTRAINT_ERROR when a value exceeds SYSTEM.MAX_INT. This implementation raises NUMERIC_ERROR during execution. (See test E24201A.)

d. Expression evaluation.

The order in which expressions are evaluated and the time at which constraints are checked are not defined by the language. While the ACVC tests do not specifically attempt to determine the order of evaluation of expressions, test results indicate the following:

(1) None of the default initialization expressions for record components are evaluated before any value is checked for membership in a component's subtype. (See test C32117A.)

(2) Assignments for subtypes are performed with the same precision as the base type. (See test C35712B.)

(3) This implementation uses no extra bits for extra precision and uses all extra bits for extra range. (See test C35903A.)

(4) NUMERIC_ERROR is raised when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)

(5) NUMERIC_ERROR is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type. (See test C45252A.)

(6) Underflow is gradual. (See tests C45524A..Z (26 tests).)

e. Rounding.

The method by which values are rounded in type conversions is not defined by the language. While the ACVC tests do not specifically attempt to determine the method of rounding, the test results indicate the following.

(1) The method used for rounding to integer is round to even. (See tests C46012A..Z (26 tests).)

(2) The method used for rounding to longest integer is round to even. (See tests C46012A..Z (26 tests).)

(3) The method used for rounding to integer in static universal real expressions is round to even. (See test C4A014A.)

f. Array types.

An implementation is allowed to raise NUMERIC_ERROR or CONSTRAINT_ERROR for an array having a 'LENGTH that exceeds STANDARD.INTEGER'LAST and/or SYSTEM.MAX_INT. For this implementation:

(1) Declaration of an array type or subtype declaration with more than SYSTEM.MAX_INT components raises NUMERIC_ERROR. (See test C36003A.)

(2) CONSTRAINT_ERROR is raised when an array type with INTEGER'LAST + 2 components is declared. (See test C36202A.)

(3) CONSTRAINT_ERROR is raised when an array type with SYSTEM.MAX_INT + 2 components is declared. (See test C36202B.)

(4) A packed BOOLEAN array having a 'LENGTH exceeding INTEGER'LAST raises CONSTRAINT_ERROR when the array type is declared. (See test C52103X.)

(5) A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises CONSTRAINT_ERROR when the array type is declared. (See test C52104Y.)

(6) In assigning one-dimensional array types, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

(7) In assigning two-dimensional array types, the expression is not evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

g. A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises CONSTRAINT_ERROR when the array type is declared. (See test E52103Y.)

h. Discriminated types.

(1) In assigning record types with discriminants, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

i. Aggregates.

(1) In the evaluation of a multi-dimensional aggregate, the test results indicate that all choices are evaluated before checking against the index type. (See tests C43207A and C43207B.)

(2) In the evaluation of an aggregate containing subaggregates, not all choices are evaluated before being checked for identical bounds. (See test E43212B.)

(3) CONSTRAINT_ERROR is raised after all choices are evaluated when a bound in a non-null range of a non-null aggregate does not belong to an index subtype. (See test E43211B.)

j. Pragmas.

(1) The pragma INLINE is supported for functions or procedures calls within a body. The program INLINE for function calls with a declarative part is not supported. (See tests LA3004A..B (2 tests), EA3004C..D (2 tests), and CA3004E..F (2 tests).)

k. Generics.

(1) Generic specifications and bodies can be compiled in separate compilations. (See tests CA1012A, CA2009C, CA2009F, BC3204C, and BC3205D.)

(2) Generic subprogram declarations and bodies can be compiled in separate compilations. (See tests CA1012A and CA2009F.)

(3) Generic library subprogram specifications and bodies can be compiled in separate compilations. (See test CA1012A.)

(4) Generic non-library package bodies as subunits can be compiled in separate compilations. (See test CA2009C.)

(5) Generic non-library subprogram bodies can be compiled in separate compilations from their stubs. (See test CA2009F.)

(6) Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

(7) Generic package declarations and bodies can be compiled in separate compilations. (See tests CA2009C, BC3204C, and BC3205D.)

(8) Generic library package specifications and bodies can be compiled in separate compilations. (See tests BC3204C and BC3205D.)

l. Input and output.

(1) The package SEQUENTIAL_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101C, EE2201D and EE2201E.)

(2) The package DIRECT_IO can be instantiated with unconstrained array types and record types with discriminants without defaults. (See tests AE2101H, EE2401D and EE2401G.)

(3) Modes IN_FILE and OUT_FILE are supported for SEQUENTIAL_IO. (See tests CE2102D..E, CE2102N and CE2102P.)

(4) Modes IN_FILE, OUT_FILE, and INOUT_FILE are supported for DIRECT_IO. (See tests CE2102F, CE2102I..J (2 tests), CE2102R, CE2102T and CE2102V.)

(5) Modes IN_FILE and OUT_FILE are supported for text files. (See tests CE3102E and CE3102I..K (3 tests).)

(6) RESET and DELETE operations are supported for SEQUENTIAL_IO. (See tests CE2102G and CE2102X.)

(7) RESET and DELETE operations are supported for DIRECT_IO. (See tests CE2102K and CE2102Y.)

(8) RESET and DELETE operations are supported for text files. (See tests CE3102F..G (2 tests), CE3104C, CE3110A and CE3114A.)

(9) Overwriting to a sequential file truncates to the last element written. (See test CE2208B.)

(10) Temporary sequential files are given names and deleted when closed. (See test CE2108A.)

(11) Temporary direct files are given names and deleted when closed. (See test CE2108C.)

(12) Temporary text files are given names and deleted when closed (See test CE3112A.)

(13) More than one internal file can be associated with each external file for sequential files when reading only. (See tests CE2107A..E (5 tests), CE2102L, CE2110B and CE2111D.)

(14) More than one internal file can be associated with each external file for direct files when reading only. (See tests CE2107F..H (3 tests), CE2110D and CE2111H.)

(15) More than one internal file can be associated with each external file for text files when reading only. (See tests CE3111A..E (5 tests), CE3114B and CE3115A.)

CHAPTER 3

TEST INFORMATION

3.1 TEST RESULTS

Version 1.10 of the ACVC comprises 3717 tests. When this compiler was tested, 44 tests had been withdrawn because of test errors. The AVF determined that 390 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 201 executable tests that use floating-point precision exceeding that supported by the implementation. Modifications to the code, processing, or grading for 42 tests were required to successfully demonstrate the test objective. (See section 3.6.)

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	129	1131	1934	17	26	46	3283
Inapplicable	0	7	381	0	2	0	390
Withdrawn	1	2	35	0	6	0	44
TOTAL	130	1140	2350	17	34	46	3717

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14	
Passed	198	577	544	245	172	99	160	332	137	36	252	251	280	3283
Inapp	14	72	136	3	0	0	6	0	0	0	0	118	41	390
Withdrawn	1	1	0	0	0	0	0	2	0	0	1	35	4	44
TOTAL	213	650	680	248	172	99	166	334	137	36	253	404	325	3717

3.4 WITHDRAWN TESTS

The following 44 tests were withdrawn from ACVC Version 1.10 at the time of this validation:

E28005C	A39005G	B97102E	C97116A
BC3009B	CD2A62D	CD2A63A..D (4 tests)	CD2A66A..D (4 tests)
CD2A73A..D (4 tests)	CD2A76A..D (4 tests)	CD2A81G	CD2A83G
CD2A84M..N (2 tests)	CD5011O	CD2B15C	CD7205C
CD2D11B	CD5007B	ED7004B	ED7005C..D (2 tests)
ED7006C..D (2 tests)	CD7105A	CD7203B	CD7204B
CD7205D	CE2107I	CE3111C	CE3301A
CE3411B			

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 390 tests were inapplicable for the reasons indicated:

a. The following 201 tests are not applicable because they have floating-point type declarations requiring more digits than SYSTEM.MAX_DIGITS:

C24113L..Y (14 tests)	C35705L..Y (14 tests)	C35706L..Y (14 tests)
C35707L..Y (14 tests)	C35708L..Y (14 tests)	C35802L..Z (15 tests)
C45241L..Y (14 tests)	C45321L..Y (14 tests)	C45421L..Y (14 tests)
C45521L..Z (15 tests)	C45524L..Z (15 tests)	C45621L..Z (15 tests)
C45641L..Y (14 tests)	C46012L..Z (15 tests)	

b. C35702A and B86001T are not applicable because this implementation supports no predefined type SHORT_FLOAT.

c. The following 16 tests are not applicable because this implementation does not support a predefined type LONG_INTEGER:

C45231C	C45304C	C45502C	C45503C	C45504C
C45504F	C45611C	C45613C	C45614C	C45631C
C45632C	B52004D	C55B07A	B55B09C	B86001W
CD7101F				

d. C45531M..P (4 tests) and C45532M..P (4 tests) are inapplicable because the size of a mantissa of a fixed point type is limited to 31 bits.

e. C86001F is not applicable because, for this implementation, the package TEXT_IO is dependent upon package SYSTEM. These tests recompile package SYSTEM, making package TEXT_IO, and hence package REPORT, obsolete.

f. B86001X, C45231D, and CD7101G are not applicable because this implementation does not support any predefined integer type with a name other than INTEGER, LONG_INTEGER, or SHORT_INTEGER.

g. B86001Y is not applicable because this implementation supports no predefined fixed-point type other than DURATION.

h. B86001Z is not applicable because this implementation supports no predefined floating-point type with a name other than FLOAT, LONG_FLOAT, or SHORT_FLOAT.

i. CD1009C, CD2A41A..E (5 tests) and CD2A42A..J (10 tests) are not applicable because the SIZE clause on type FLOAT is not supported by this implementation.

j. The following 26 tests are inapplicable because for this implementation a length clause on a type derived from a private type is not supported outside the defined package.

CD1C04A	CD2A21C..D (2 tests)	CD2A22C..D (2 tests)
CD2A22G..H (2 tests)	CD2A31C..D (2 tests)	CD2A32C..D (2 tests)
CD2A32G..H (2 tests)	CD2A51C..D (2 tests)	CD2A52C..D (2 tests)
CD2A52G..H (2 tests)	CD2A53D	CD2A54D
CD2A54H	CD2A72A..B (2 tests)	CD2A75A..B (2 tests)

k. CD1C04B, CD1C04E, CD4051A..B (2 tests) and CD4051C..D (2 tests) are not applicable because this implementation does not support representation clauses in derived records or derived tasks.

l. The following 25 tests are inapplicable because a LENGTH clause on an array or record would require a change to the representation of the components or elements.

CD2A61A..D (4 tests)	CD2A61F	CD2A61H..L (5 tests)
CD2A62A..C (3 tests)	CD2A71A..D (4 tests)	CD2A72C..D (2 tests)
CD2A74A..D (4 tests)	CD2A75C..D (2 tests)	

m. CD2A84B..I (8 tests) and CD2A84K..L (2 tests) are not applicable because the 'SIZE clause applied to the access type is less than the minimum (32 bits) required.

n. The following 30 tests are inapplicable because an ADDRESS clause for a constant is not supported.

CD5011B	CD5011D	CD5011F
CD5011H	CD5011L	CD5011N
CD5011R..S (2 tests)	CD5012C..D (2 tests)	CD5012G..H (2 tests)

CD5012L	CD5013B	CD5013D
CD5013F	CD5013H	CD5013L
CD5013N	CD5013R	CD5014B
CD5014D	CD5014F	CD5014H
CD5014J	CD5014L	CD5014N
CD5014R	CD5014U	CD5014W

o. CD5012J, CD5013S and CD5014S are not applicable because ADDRESS clauses for tasks are not supported.

p. CE2102D is inapplicable because this implementation supports CREATE with IN_FILE mode for SEQUENTIAL_IO.

q. CE2102E is inapplicable because this implementation supports CREATE with OUT_FILE mode for SEQUENTIAL_IO.

r. CE2102F is inapplicable because this implementation supports CREATE with INOUT_FILE mode for DIRECT_IO.

s. CE2102I is inapplicable because this implementation supports CREATE with IN_FILE mode for DIRECT_IO.

t. CE2102J is inapplicable because this implementation supports CREATE with OUT_FILE mode for DIRECT_IO.

u. CE2102N is inapplicable because this implementation supports OPEN with IN_FILE mode for SEQUENTIAL_IO.

v. CE2102O is inapplicable because this implementation supports RESET with IN_FILE mode for SEQUENTIAL_IO.

w. CE2102P is inapplicable because this implementation supports OPEN with OUT_FILE mode for SEQUENTIAL_IO.

x. CE2102Q is inapplicable because this implementation supports RESET with OUT_FILE mode for SEQUENTIAL_IO.

y. CE2102R is inapplicable because this implementation supports OPEN with INOUT_FILE mode for DIRECT_IO.

z. CE2102S is inapplicable because this implementation supports RESET with INOUT_FILE mode for DIRECT_IO.

aa. CE2102T is inapplicable because this implementation supports OPEN with IN_FILE mode for DIRECT_IO.

ab. CE2102U is inapplicable because this implementation supports RESET with IN_FILE mode for DIRECT_IO.

ac. CE2102V is inapplicable because this implementation supports OPEN with OUT_FILE mode for DIRECT_IO.

ad. CE2102W is inapplicable because this implementation supports RESET with OUT_FILE mode for DIRECT_IO.

ae. CE2107B..E (4 tests), CE2107L, CE2110B and CE2401H are not applicable because multiple internal files cannot be associated with the same external file when one or more files is writing for sequential files. The proper exception is raised when multiple access is attempted.

af. CE2107G..H (2 tests), CE2110D, CE2111D and CE2111H are not applicable because multiple internal files cannot be associated with the same external file when one or more files is writing for direct files. The proper exception is raised when multiple access is attempted.

ag. EE2401D, and EE2401G use instantiations of package DIRECT_IO with unconstrained array types and record types with discriminants with defaults. These instantiations cause USE_ERROR to be raised without a FORM parameter.

ah. CE3102E is inapplicable because text file CREATE with IN_FILE mode is supported by this implementation.

ai. CE3102F is inapplicable because text file RESET is supported by this implementation.

aj. CE3102G is inapplicable because text file deletion of an external file is supported by this implementation.

ak. CE3102I is inapplicable because text file CREATE with OUT_FILE mode is supported by this implementation.

al. CE3102J is inapplicable because text file OPEN with IN_FILE mode is supported by this implementation.

am. CE3102K is inapplicable because text file OPEN with OUT_FILE mode is not supported by this implementation.

an. CE3111B, CE3111D..E (2 tests), CE3114B, and CE3115A are not applicable because multiple internal files cannot be associated with the same external file when one or more files is writing for text files. The proper exception is raised when multiple access is attempted.

ao. CE3202A requires association of a name with the standard input/output files, but this is not supported by this implementation which raises USE_ERROR. This behaviour is accepted by the AVO pending a ruling by the language maintenance body.

ap. CE3605A is inapplicable because this test attempts to output a string of 517 characters which exceeds the maximum allowed for this implementation.

3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made by the AVF in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into subtests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that was not anticipated by the test (such as raising one exception instead of another).

Modifications were required for **42** tests.

The following tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

B23004A	B24007A	B24009A	B28003A	B28003C
B32202A	B32202B	B32202C	B33001A	B37004A
B45102A	B61012A	B62001B	B62001C	B62001D
B74304A	B74401F	B74401R	B91004A	B95069A
B95069B	B97103E	BA1101B2	BA1101B4	BC2001D
BC3009C	BD5005B			

The following tests were split to prove the not-applicability criteria:

CD2A62A	CD2A62B	CD2A72A	CD2A72B	CD2A75A
CD2A75B	CD2A84B	CD2A84C	CD2A84D	CD2A84E
CD2A84F	CD2A84G	CD2A84H	CD2A84I	

EA3004D, when processed, produces only two of the expected three errors: the implementation fails to detect an error on line 27 of file EA3004D6M. This is because the pragma INLINE has no effect when its object is within a package specification. The task was reordered to compile files D2 and D3 after file D5 (the re-compilation of the "with"ed package that makes the various earlier units obsolete), the re-ordered test executed and produced the expected NOT_APPLICABLE result (as though INLINE were not supported at all). The re-ordering of EA3004D test files was: 0-1-4-5-2-3-6. The AVO ruled that the test should be counted as passed.

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.10 produced by the **AlsyCOMP_017 V4.0** compiler was submitted to the AVF by the applicant for review. Analysis f these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the **AlsyCOMP_017 V4.0** compiler using ACVC Version 1.10 was conducted on-site by a validation team from the AVF. The configuration in which the testing was performed is described by the following designations of hardware and software components:

Host computer	: **MicroVAX II**
Host operating system	: **MicroVMS V4.7**
Target computer	: **INMOS T800 implemented on a B403 TRAM (bare), using the Host running INMOS Iserver V1.30 for file-server support.**
Compiler	: **AlsyCOMP_017 V4.0**
Pre-linker	: **AlsyCOMP_017 V4.0**
Linker	: **IMS D605A ILINK V2.0.2**
Loader/Downloader	: **IMS D605A IBOOT V1.0.3**
Runtime System	: **AlsyCOMP_017 V4.0**

The host and target computers were linked via **CAPLIN QT0 Board.**

A magnetic tape containing all tests except for withdrawn tests and tests requiring unsupported floating-point precisions was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized before being written to the magnetic tape. Tests requiring modifications during the prevalidation testing were not included in their modified form on the magnetic tape.

The contents of the magnetic tape were not loaded directly onto the host computer, but loaded on to a hard disc via a VAX 11/780. The disc was then manually swiched to allow the MicroVAX to access the test files.

After the test files were loaded to disk, the full set of tests was compiled and linked on the MicroVAX, then all executable images were transferred to the INMOS T800 via the CAPLINK and run. Results were transfered from the host computer to the VAX 11/750 via FTP software from where they were printed.

The compiler was tested using command scripts provided by **Alsys Limited** and reviewed by the validation team. The compiler was tested using all the following default option settings:

OPTION	EFFECT

CALLS=INLINE	Allows inline insertion of code for subprograms.
OBJECT=NONE	No peephole optimisations are performed, this is done for compilation speed improvements.
OUTPUT=<file>	<file> specifies the name of compilation listing generated.

In addition the following options were used to produce full compiler listings:

TEXT	Print a compilation listing including full source text.
SHOW=NONE	Do not print a header and do not include an error summary in the compilation listing.
ERROR=999	Set the maximum number of compilation errors permitted before compilation is terminated to 999.
MONITOR_WIDTH=80	Set width for standard output to 80 columns.
FILE_WIDTH=80	Set width for listing file to 80 columns.
FILE_LENGTH=9999	Disable insertion of form feeds in the output.

Tests were compiled, linked, and executed (as appropriate) using a single host and target computer. Test output, compilation listings, and job logs were captured on magnetic media and archived at the AVF. The listings examined on-site by the validation team were also archived.

3.7.3 Test Site

Testing was conducted at **Alsys Limited, Partridge House, Newtown Road, Henley-on-Thames, Oxfordshire, RG9 1EN, UNITED KINGDOM** and was completed on **31 May 1989**.

APPENDIX A

DECLARATION OF CONFORMANCE

Alsys Limited has submitted the following Declaration of Conformance concerning the **AlsyCOMP_017 V4.0** compiler.

DECLARATION OF CONFORMANCE

Compiler Implementor: **Alsys Limited**

Ada Validation Facility: **The National Computing Centre Limited,**
Oxford Road
Manchester
M1 7ED
UNITED KINGDOM

Ada Compiler Validation Capability (ACVC) Version: **1.10**

Base Configuration

Base Compiler Name: **AlsyCOMP_017 V4.0**

Host Architecture: **MicroVAX II**
Host OS and Version: **MicroVMS V4.7**

Target Architecture: **INMOS T800 transputer implemented on a B403 TRAM (bare), using the Host running INMOS Iserver V1.30 for file-server support via a CAPLIN QT0 board link**

Implementor's Declaration

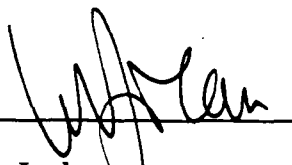
I, the undersigned, representing **Alsys Limited**, have implemented no deliberate extensions to the Ada Language Standard ANSI/MIL-STD-1815A in the compiler(s) listed in this declaration. I declare that **Alsys Limited** is the owner of record of the Ada language compiler(s) listed above and, as such, is responsible for maintaining said compiler(s) in conformance to ANSI/MIL-STD-1815A. All certificates and registrations for Ada language compiler(s) listed in this declaration shall be made only in the owner's corporate name.

Date : 21/6/89

Martyn Jordan
Marketing Director

Owner's Declaration

I, the undersigned, representing **Alsys Limited**, take full responsibility for implementation and maintenance of the Ada compiler(s) listed above, and agree to the public disclosure of the final Validation Summary Report. I declare that all of the Ada language compilers listed, and their host/target performance, are in compliance with the Ada Language Standard ANSI/MIL-STD-1815A.



Date : 21/6/89

Martyn Jordan
Marketing Director

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of the **AlsysCOMP_017 V4.0** compiler, as described in this Appendix, are provided by **Alsys Limited**. Unless specifically noted otherwise, references in this appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

```
package STANDARD is

...

type INTEGER is range -2**31 .. 2**31-1;
type SHORT_INTEGER is range -2**07 .. 2**07-1;

type FLOAT is digits 6 range -(2.0 - 2.0**(-23)) * 2.0**127 .. (2.0 - 2.0**(-23)) * 2.0**127;

type LONG_FLOAT is digits 15 range -(2.0 - 2.0**(-51)) * 2.0**1023 ..
                                    (2.0 - 2.0**(-51)) * 2.0**1023;

type DURATION is delta 2.0**-14 range -86400.0 .. 86400.0;

...

end STANDARD;
```

Alsys transputer Ada Compiler

APPENDIX F

Implementation - Dependent Characteristics

Version 4.0

Alsys S.A.
29, Avenue de Versailles
78170 La Celle St. Cloud, France

Alsys Inc.
67 South Bedford Street
Burlington, MA 01803-5152, U.S.A.

Alsys Ltd.
Partridge House, Newtown Road
Henley-on-Thames,
Oxfordshire RG9 1EN, U.K.

PREFACE

This *Alsys transputer Ada Compiler Appendix F* is for programmers, software engineers, project managers, educators and students who want to develop an Ada program for the INMOS transputer.

This appendix is a required part of the *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD 1815A, January 1983 (throughout this appendix, citations in square brackets refer to this manual).

This document assumes that the reader has some knowledge of the architecture of the transputer. Access to the document *OCCAM2 toolset manual (72-TDS-184-00)*, which describes INMOS provided OCCAM programming tools, would also be advantageous.

TABLE OF CONTENTS

APPENDIX F 1

1 Implementation-Dependent Pragmas 2
1.1 INLINE 2
1.2 INTERFACE 2
1.2.1 Calling Conventions 2
1.2.2 Parameter-Passing Conventions 3
1.2.3 Parameter Representations 3
1.2.4 Restrictions on Interfaced Subprograms 5
1.3 INTERFACE_NAME 6
1.4 Other Pragmas 6

2 Implementation-Dependent Attributes 7

3 Specification of the Package SYSTEM 8

4 Restrictions on Representation Clauses 9
4.1 Enumeration Types 9
4.2 Integer Types 12
4.3 Floating Point Types 14
4.4 Fixed Point Types 16
4.5 Access Types 19
4.6 Task Types 20
4.7 Array Types 21
4.8 Record Types 24

5 Conventions for Implementation-Generated Names 33

6 Address Clauses 34
6.1 Address Clauses for Objects 34
6.2 Address Clauses for Program Units 34
6.3 Address Clauses for Entries 34

7 Restrictions on Unchecked Conversions 35

8 Input-Output Packages 36
8.1 NAME Parameter 36
8.2 FORM Parameter 36
8.3 USE_ERROR 38

9 Characteristics of Numeric Types 40
9.1 Integer Types - T2 transputer targets 40
9.2 Integer Types - T4/T8 transputer targets 40
9.3 Other Integer Types 40
9.4 Floating Point Type Attributes 41
9.5 Attributes of Type DURATION 42

INDEX 43

APPENDIX F

Implementation-Dependent Characteristics

This appendix summarizes the implementation-dependent characteristics of the Alsys Ada Compilers for the INMOS transputer. This document should be considered as the Appendix F to the Reference Manual for the Ada Programming Language ANSI/MIL-STD 1815A, January 1983, as appropriate to the Alsys Ada implementation for the transputer.

Sections 1 to 8 of this appendix correspond to the various items of information required in Appendix F [F]*; section 9 provides other information relevant to the Alsys implementation. The contents of all these sections is described below:

1. The form, allowed places, and effect of every implementation-dependent pragma.

2. The name and type of every implementation-dependent attribute.

3. The specification of the package SYSTEM [13.7].

4. The list of all restrictions on representation clauses [13.1].

5. The conventions used for any implementation-generated names denoting implementation-dependent components [13.4].

6. The interpretation of expressions that appear in address clauses, including those for interrupts [13.5].

7. Any restrictions on unchecked conversions [13.10.2].

8. Any implementation-dependent characteristics of the input-output packages [14].

9. Characteristics of numeric types.

Throughout this appendix, the name *Ada Run-Time Executive* refers to the run-time library routines provided for all Ada programs. These routines implement the Ada heap, exceptions, tasking control, I/O, and other utility functions.

---

* Throughout this manual, citations in square brackets refer to the *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A, January 1983.

1 Implementation-Dependent Pragmas

1.1 INLINE

Pragma INLINE [6.3.2] is fully supported, except for the fact that it is not possible to inline a function call in a declarative part.

1.2 INTERFACE

Ada programs can interface to subprograms written in OCCAM through the use of the predefined pragma INTERFACE [13.9] and the implementation-defined pragma INTERFACE_NAME.

Pragma INTERFACE specifies the name of an interfaced subprogram and the name of the programming language for which calling and parameter passing conventions will be generated. Pragma INTERFACE takes the form specified in the *Reference Manual*:

pragma INTERFACE (*language_name*, *subprogram_name*);

where:

- *language_name* is the name of the other language whose calling and parameter passing conventions are to be used.
- *subprogram_name* is the name used within the Ada program to refer to the interfaced subprogram.

The only language name currently accepted by pragma INTERFACE is OCCAM.

The language name used in the pragma INTERFACE does not necessarily correspond to the language used to write the interfaced subprogram. It is used only to tell the Compiler how to generate subprogram calls, that is, which calling conventions and parameter passing techniques to use.

The language name OCCAM is used to refer to the standard OCCAM calling and parameter passing conventions for the transputer. The programmer can use the language name OCCAM to interface Ada subprograms with subroutines written in any language that follows the standard OCCAM calling conventions.

1.2.1 Calling Conventions

The following calling conventions are required for code to be interfaced to Ada by use of the pragma interface to OCCAM.

On entry to the subprogram, the registers A, B and C are undefined. For the T8 only, the floating point registers FA, FB and FC are similarly undefined. The return address and any parameters are accessed relative to the workspace pointer, W.

There are no assumptions concerning the register contents upon return from the interfaced subprogram, other than for interfaced subprograms which are functions (see below).

1.2.2 Parameter-Passing Conventions

On entry to the subprogram, the first word above the transputer workspace pointer contains the return address of the called subprogram. Subsequent workspace locations (from W+1 to W+n, where n is the number of parameters) contain the subprograms parameters, which are all one word in length.

There is always an implicit *vector space* parameter passed as the last parameter to all interfaced subprograms. This points to an area of free memory for use by the OCCAM compiler in allocating arrays declared in the interfaced subprogram.

Formal parameters of mode **in** which are access types or scalars of one machine word or less in size are passed by copy. If such a parameter is less that one machine word in length it is sign extended to a full word. For all other parameters the value passed is the address of the actual parameter itself.

When passing arrays to OCCAM, it may be the case that some of its strides (dimensions) are undefined in the source of the interfaced subprogram. If this is true, the missing strides should be passed as extra integer value parameters to the subprogram. These parameters should be placed immediately following the array parameter itself and in the same order as the missing strides appear in the OCCAM source.

Since all large scalar, non-scalar and non-access parameters to interfaced subprograms are passed by address, they cannot be protected from modification by the called subprogram even though they may be formally declared to be of mode **in**. It is the programmer's responsibility to ensure that the semantics of the Ada parameter modes are honoured in these cases.

If the subprogram is a function whose result is at most one machine word in length, register A is used to return the result. All other results are returned by address in an implicit parameter allocated before the list of normal parameters (i.e. in the first word after the return address, at W+1).

No consistency checking is performed between the subprogram parameters declared in Ada and the corresponding parameters of the interfaced subprogram. It is the programmer's responsibility to ensure correct access to the parameters.

1.2.3 Parameter Representations

This section describes the representation of values of the types that can be passed as parameters to an interfaced subprogram. The discussion assumes no representation clauses have been used to alter the default representations of the types involved. Chapter 4 describes the effect of representation clauses on the representation of values.

Integer Types [3.5.4]

Ada integer types are represented in two's complement form and occupy a byte (SHORT_INTEGER), a word (INTEGER) or a double word (LONG_INTEGER). Parameters to interfaced subprograms of type SHORT_INTEGER are passed by copy in a full machine word. The value occupies the low order byte of the word; the other bytes in the word are always zero. Values of type INTEGER are always passed by copy. The predefined type LONG_INTEGER is available for T2 transputer targets only; values of this type are stored least significant word first and are always passed by address.

Enumeration Types [3.5.1]

Values of an Ada enumeration type are represented internally as unsigned values representing their position in the list of enumeration literals defining the type. The first literal in the list corresponds to a value of zero.

Enumeration types with 256 elements or fewer are represented in 8 bits. For T2 transputer targets, those with between 257 and 65536 (2**16) elements are represented in 16 bits (i.e. a word). All others enumeration types are represented in 32 bits. The maximum number of values an enumeration type can include is 2**31.

Consequently, the Ada predefined type CHARACTER [3.5.2] is represented in 8 bits, using the standard ASCII codes [C] and the Ada predefined type BOOLEAN [3.5.3] is represented in 8 bits, with FALSE represented by the value 0, and TRUE represented by the value 1.

As the representation of enumeration types is basically the same that of integers, the same parameter passing conventions apply.

Floating Point Types [3.5.7, 3.5.8]

Ada floating-point values occupy 32 (FLOAT) or 64 (LONG_FLOAT) bits, and are held in ANSI/IEEE 754 floating point format.

Fixed Point Types [3.5.9, 3.5.10]

Ada fixed-point types are managed by the Compiler as the product of a signed *mantissa* and a constant *small*. The mantissa is implemented as an 8, 16 or 32 bit integer value for T2 transputer targets and as an 8 or 32 bit integer value for T4 and T8 transputer targets. *Small* is a compile-time quantity which is the power of two equal or immediately inferior to the delta specified in the declaration of the type.

The attribute MANTISSA is defined as the smallest number such that:

2 ** MANTISSA >= max (abs (upper_bound), abs (lower_bound)) / *small*

The size of a fixed point type is:

MANTISSA	Size

1 .. 7	8 bits	
1 .. 15	16 bits	(T2 transputer targets only).
16 .. 31	32 bits	

Fixed point types requiring a MANTISSA greater than 31 are not supported.

Access Types [3.8]

Values of access types are represented internally by the address of the designated object held in single word. The value MIN_INT (the smallest integer that can be represented in a machine word) is used to represent **null**.

Array Types [3.6]

Ada arrays are passed by address; the value passed is the address of the first element of the first dimension of the array. The elements of the array are allocated by row. When an array is passed as a parameter to an interfaced subprogram, the usual consistency checking between the array bounds declared in the calling program and the subprogram is not enforced. It is the programmer's responsibility to ensure that the subprogram does not violate the bounds of the array.

Values of the predefined type STRING [3.6.3] are arrays, and are passed in the same way: the address of the first character in the string is passed. Elements of a string are represented in 8 bits, using the standard ASCII codes.

Record Types [3.7]

Ada records are passed by address; the value passed is the address of the first component of the record. Components of a record are aligned on their natural boundaries (e.g. INTEGER on a word boundary) and the components may be re-ordered by the Compiler so as to minimize the total size of objects of the record type. If a record contains discriminants or components having a dynamic size, implicit components may be added to the record. Thus the default layout of the internal structure of the record may not be inferred directly from its Ada declaration. The use of a representation clause to control the layout of any record type whose values are to be passed to interfaced subprograms is recommended.

1.2.4 Restrictions on Interfaced Subprograms

Interfaced OCCAM subprograms must be compiled using the UNIVERSAL error mode (X). In this mode, there is no error checking and any run-time errors in the OCCAM code are ignored. This ensures that processes do not execute a STOPP instruction and avoids the unpredictable results which may occur if this is allowed to happen.

It is not possible to interface to OCCAM functions which return floating point types, nor to those which have more that one return value.

1.3 INTERFACE_NAME

Pragma INTERFACE_NAME associates the name of an interfaced subprogram, as declared in Ada, with its name in the language of origin. If pragma INTERFACE_NAME is not used, then the two names are assumed to be identical.

This pragma takes the form:

pragma INTERFACE_NAME (*subprogram_name*, *string_literal*);

where:

- *subprogram_name* is the name used within the Ada program to refer to the interfaced subprogram.
- *string_literal* is the name by which the interfaced subprogram is referred to at link-time.

The use of INTERFACE_NAME is optional, and is not needed if a subprogram has the same name in Ada as in the language of origin. It is necessary, for example, if the name of the subprogram in its original language contains characters that are not permitted in Ada identifiers. Ada identifiers can contain only letters, digits and underscores, whereas the INMOS linker allows external names to contain other characters, e.g. full stops. These characters can be specified in the *string_literal* argument of the pragma INTERFACE_NAME.

The pragma INTERFACE_NAME is allowed at the same places of an Ada program as the pragma INTERFACE [13.9]. However, the pragma INTERFACE_NAME must always occur after the pragma INTERFACE declaration for the interfaced subprogram.

1.4 Other Pragmas

Pragmas IMPROVE and PACK are discussed in detail in the section on representation clauses (Chapter 4).

Pragma PRIORITY is accepted with the range of priorities running from 1 to 10 (see the definition of the predefined package SYSTEM in Chapter 3). The undefined priority (no pragma PRIORITY) is treated as though it were less than any defined priority value.

In addition to pragma SUPPRESS, it is possible to suppress all checks in a given compilation by the use of the Compiler option CHECKS.

The following language defined pragmas have no effect.

CONTROLLED
MEMORY_SIZE
OPTIMIZE
STORAGE_UNIT
SYSTEM_NAME

Note that all access types are implemented by default as controlled collections as described in [4.8].

2 Implementation-Dependent Attributes

In addition to the Representation Attributes of [13.7.2] and [13.7.3], the four attributes listed in section 5 (Conventions for Implementation-Generated Names), for use in record representation clauses, and the attributes described below are provided:

T'DESCRIPTOR_SIZE	For a prefix T that denotes a type or subtype, this attribute yields the size (in bits) required to hold a descriptor for an object of the type T, allocated on the heap or written to a file. If T is constrained, T'DESCRIPTOR_SIZE will yield the value 0.
T'IS_ARRAY	For a prefix T that denotes a type or subtype, this attribute yields the value TRUE if T denotes an array type or an array subtype; otherwise, it yields the value FALSE.

Limitations on the use of the attribute ADDRESS

The attribute ADDRESS is implemented for all prefixes that have meaningful addresses. The following entities do not have meaningful addresses and will therefore cause a compilation error if used as a prefix to ADDRESS:

- A constant or named number that is implemented as an immediate value (i.e. does not have any space allocated for it).
- A package specification that is not a library unit.
- A package body that is not a library unit or subunit.

3 Specification of the Package SYSTEM

```
package SYSTEM is

   type NAME is (TRANSPUTER);

   SYSTEM_NAME : constant NAME := NAME'FIRST;
   MIN_INT     : constant := -(2**31);
   MAX_INT     : constant := 2**31-1;
   MEMORY_SIZE : constant := 2**16;                -- for T2 transputer targets
   MEMORY_SIZE : constant := 2**31-1;              -- for T4/T8 transputer targets

   type ADDRESS is new INTEGER;

   STORAGE_UNIT : constant := 8;
   MAX_DIGITS   : constant := 15;
   MAX_MANTISSA : constant := 31;
   FINE_DELTA   : constant := 2#1.0#e-31;
   TICK         : constant := 1.0e-6;
   NULL_ADDRESS : constant ADDRESS := ADDRESS'FIRST;

   subtype PRIORITY is INTEGER range 1 .. 10;

end SYSTEM;
```

4 Restrictions on Representation Clauses

This section explains how objects are represented and allocated by the Alsys transputer Ada Compiler and how it is possible to control this using representation clauses.

The representation of an object is closely connected with its type. For this reason this section addresses successively the representation of enumeration, integer, floating point, fixed point, access, task, array and record types. For each class of type the representation of the corresponding objects is described.

The transputer supports operations on the data types byte, word and double-word, so these data types are used to form the basis of the representation of Ada types. The word length for T4 and T8 transputer targets is 32 bits whereas T2 transputers have a word length of only 16 bits. Currently, the compiler does not support operations on double 32 bit word quantities. This affects the representation of integer, fixed point and enumeration types.

Except in the case of array and record types, the description of each class of type is independent of the others. To understand the representation of an array type it is necessary to understand first the representation of its components. The same rule applies to a record type.

Apart from implementation defined pragmas, Ada provides three means to control the size of objects:

- a (predefined) pragma PACK, when the object is an array, an array component, a record or a record component
- a record representation clause, when the object is a record or a record component
- a size specification, in any case.

For each class of types the effect of a size specification is described. Interaction between size specifications, packing and record representation clauses is described under array and record types.

Representation clauses on derived record types or derived task types are not supported.

Size representation clauses on types derived from private types are not supported when the derived type is declared outside the private part of the defining package.

4.1 Enumeration Types

Internal codes of enumeration literals

When no enumeration representation clause applies to an enumeration type, the internal code associated with an enumeration literal is the position number of the enumeration

literal. Then, for an enumeration type with n elements, the internal codes are the integers 0, 1, 2, ... , n-1.

An enumeration representation clause can be provided to specify the value of each internal code as described in [13.3]. The Alsys Compiler fully implements enumeration representation clauses.

As internal codes must be machine integers the internal codes provided by an enumeration representation clause must be in the range -2^{31} .. $2^{31}-1$.

Encoding of enumeration values

An enumeration value is always represented by its internal code in the program generated by the Compiler.

Minimum size of an enumeration subtype

The minimum size of an enumeration subtype is the minimum number of bits that is necessary for representing the internal codes of the subtype values in normal binary form.

For a static subtype, if it has a null range its minimum size is 1. Otherwise, if m and M are the values of the internal codes associated with the first and last enumeration values of the subtype, then its minimum size L is determined as follows. For m >= 0, L is the smallest positive integer such that M <= 2^L-1. For m < 0, L is the smallest positive integer such that -2^{L-1} <= m and M <= $2^{L-1}-1$. For example:

```
type COLOR is (GREEN, BLACK, WHITE, RED, BLUE, YELLOW);
-- The minimum size of COLOR is 3 bits.

subtype BLACK_AND_WHITE is COLOR range BLACK .. WHITE;
-- The minimum size of BLACK_AND_WHITE is 2 bits.

subtype BLACK_OR_WHITE is BLACK_AND_WHITE range X .. X;
-- Assuming that X is not static, the minimum size of BLACK_OR_WHITE is
-- 2 bits (the same as the minimum size of the static type mark
-- BLACK_AND_WHITE).
```

Size of an enumeration subtype

When no size specification is applied to an enumeration type or first named subtype, the objects of that type or first named subtype are represented as either unsigned bytes or signed words. The Compiler selects automatically the smallest such object which can hold each of the internal codes of the enumeration type (or subtype). The size of the enumeration type and of any of its subtypes is thus 8 bits in the case of an unsigned byte, or the machine word size (16 or 32 bits) in the case of a signed word.

When a size specification is applied to an enumeration type, this enumeration type and each of its subtypes has the size specified by the length clause. The same rule applies to a first named subtype. The size specification must of course specify a value greater than or equal to the minimum size of the type or subtype to which it applies. For example:

```
type EXTENDED is
  (     -- The usual American ASCII characters.
     NUL,   SOH,   STX,   ETX,   EOT,   ENQ,   ACK,   BEL,
     BS,    HT,    LF,    VT,    FF,    CR,    SO,    SI,
     DLE,   DC1,   DC2,   DC3,   DC4,   NAK,   SYN,   ETB,
     CAN,   EM,    SUB,   ESC,   FS,    GS,    RS,    US,
     ' ',   '!',   '"',   '#',   '$',   '%',   '&',   ''',
     '(',   ')',   '*',   '+',   ',',   '-',   '.',   '/',
     '0',   '1',   '2',   '3',   '4',   '5',   '6',   '7',
     '8',   '9',   ':',   ';',   '<',   '=',   '>',   '?',
     '@',   'A',   'B',   'C',   'D',   'E',   'F',   'G',
     'H',   'I',   'J',   'K',   'L',   'M',   'N',   'O',
     'P',   'Q',   'R',   'S',   'T',   'U',   'V',   'W',
     'X',   'Y',   'Z',   '[',   '\',   ']',   '^',   '_',
     '`',   'a',   'b',   'c',   'd',   'e',   'f',   'g',
     'h',   'i',   'j',   'k',   'l',   'm',   'n',   'o',
     'p',   'q',   'r',   's',   't',   'u',   'v',   'w',
     'x',   'y',   'z',   '{',   '|',   '}',   '~',   DEL,
     -- Extended characters
     LEFT_ARROW,
     RIGHT_ARROW,
     UPPER_ARROW,
     LOWER_ARROW,
     UPPER_LEFT_CORNER,
     UPPER_RIGHT_CORNER,
     LOWER_RIGHT_CORNER,
     LOWER_LEFT_CORNER,
     ...);

for EXTENDED'SIZE use 8;
-- The size of type EXTENDED will be one byte.  Its objects will be represented
-- as unsigned 8 bit values.
```

The Alsys Compiler fully implements size specifications. Nevertheless, as enumeration values are coded using integers, the specified length cannot be greater than 32 bits.

Size of the objects of an enumeration subtype

Provided its size is not constrained by a record component clause or a pragma PACK, an object of an enumeration subtype has the same size as its subtype.

Alignment of an enumeration subtype

An enumeration subtype is byte aligned if the size of the subtype is less than or equal to 8 bits, word aligned otherwise.

Address of an object of an enumeration subtype

Provided its alignment is not constrained by a record representation clause or a pragma PACK, the address of an object of an enumeration subtype is a multiple of the alignment of the corresponding subtype.

4.2 Integer Types

Predefined integer types

In the Alsys Ada implementation for the transputer the number of predefined integer types available differs depending upon the transputer target. For T4 and T8 transputer targets there are two predefined integer types:

```
type SHORT_INTEGER      is range -2**07 .. 2**07-1;
type INTEGER            is range -2**31 .. 2**31-1;
```

For T2 transputer targets there are three predefined integer types:

```
type SHORT_INTEGER      is range -2**07 .. 2**07-1;
type INTEGER            is range -2**15 .. 2**15-1;
type LONG_INTEGER       is range -2**31 .. 2**31-1;
```

Selection of the parent of an integer type

An integer type declared by a declaration of the form:

```
type T is range L .. R;
```

is implicitly derived from one of the predefined integer types. The Compiler automatically selects the predefined integer type whose range is the shortest that contains the values L to R inclusive.

Encoding of integer values

Binary code is used to represent integer values. Negative numbers are represented using two's complement.

Minimum size of an integer subtype

The minimum size of an integer subtype is the minimum number of bits that is necessary for representing the internal codes of the subtype values in normal binary form (that is to say, in an unbiased form which includes a sign bit only if the range of the subtype includes negative values).

For a static subtype, if it has a null range its minimum size is 1. Otherwise, if m and M are the lower and upper bounds of the subtype, then its minimum size L is determined as follows. For m >= 0, L is the smallest positive integer such that M <= 2^L-1. For m < 0, L is the smallest positive integer such that -2^{L-1} <= m and M <= 2^{L-1}-1. For example:

```
subtype S is INTEGER range 0 .. 7;
-- The minimum size of S is 3 bits.

subtype D is S range X .. Y;
-- Assuming that X and Y are not static, the minimum size of
-- D is 3 bits (the same as the minimum size of the static type mark S).
```

Size of an integer subtype

For T4 and T8 transputer targets, the sizes of the predefined integer types SHORT_INTEGER and INTEGER are 8 and 32 bits respectively. For T2 transputer targets, the sizes of the predefined integer types SHORT_INTEGER, INTEGER and LONG_INTEGER are 8, 16 and 32 bits respectively.

When no size specification is applied to an integer type or to its first named subtype (if any), its size and the size of any of its subtypes is the size of the predefined type from which it derives, directly or indirectly. For example:

```
type S is range 80 .. 100;
-- S is derived from SHORT_INTEGER; its size is 8 bits.

type J is range 0 .. 65535;
-- J is derived from INTEGER for T4 and T8 targets and LONG_INTEGER
-- for T2 targets; its size is 32 bits.

type N is new J range 80 .. 100;
-- N is indirectly derived from INTEGER or LONG_INTEGER as above;
-- its size is 32 bits.
```

When a size specification is applied to an integer type, this integer type and each of its subtypes has the size specified by the length clause. The same rule applies to a first named subtype. The size specification must of course specify a value greater than or equal to the minimum size of the type or subtype to which it applies, for example:

```
type S is range 80 .. 100;
for S'SIZE use 32;
-- S is derived from SHORT_INTEGER, but its size is 32 bits because
-- of the size specification.

type J is range 0 .. 255;
for J'SIZE use 8;
-- J is derived from INTEGER, but its size is 8 bits because of the
-- size specification.

type N is new J range 80 .. 100;
-- N is indirectly derived from INTEGER, but its size is 8 bits
```

```
-- because N inherits the size specification of J.
```

The Alsys Compiler implements size specifications. Nevertheless, as integers are implemented using machine integers, the specified length cannot be greater than 32 bits.

Size of the objects of an integer subtype

Provided its size is not constrained by a record component clause or a pragma PACK, an object of an integer subtype has the same size as its subtype.

Alignment of an integer subtype

An integer subtype is byte aligned if the size of the subtype is less than or equal to 8 bits, word aligned otherwise.

Address of an object of an integer subtype

Provided its alignment is not constrained by a record representation clause or a pragma PACK, the address of an object of an integer subtype is a multiple of the alignment of the corresponding subtype.

4.3 Floating Point Types

Predefined floating point types

In the Alsys Ada implementation for the transputer there are two predefined floating point types.

```
type FLOAT is
      digits 6 range  -(2.0 - 2.0**(-23))*2.0**127 .. (2.0 - 2.0**(-23))*2.0**127;

type LONG_FLOAT is
      digits 15 range  -(2.0 - 2.0**(-51))*2.0**1023 .. (2.0 - 2.0**(-51))*2.0**1023;
```

Selection of the parent of a floating point type

A floating point type declared by a declaration of the form:

```
type T is digits D [range L .. R];
```

is implicitly derived from a predefined floating point type. The Compiler automatically selects the smallest predefined floating point type whose number of digits is greater than or equal to D and which contains the values L and R.

Encoding of floating point values

In the program generated by the Compiler, floating point values are represented using the ANSI/IEEE 754 standard 32-bit and 64-bit floating point formats as appropriate.

Values of the predefined type FLOAT are represented using the 32-bit floating point format and values of the predefined type LONG_FLOAT are represented using the 64-bit floating point format as defined by the standard. The values of any other floating point type are represented in the same way as the values of the predefined type from which it derives, directly or indirectly.

Minimum size of a floating point subtype

The minimum size of a floating point subtype is 32 bits if its base type is FLOAT or a type derived from FLOAT and 64 bits if its base type is LONG_FLOAT or a type derived from LONG_FLOAT.

Size of a floating point subtype

The sizes of the predefined floating point types FLOAT and LONG_FLOAT are 32 and 64 bits respectively.

The size of a floating point type and the size of any of its subtypes is the size of the predefined type from which it derives directly or indirectly.

The only size that can be specified for a floating point type or first named subtype using a size specification is its usual size (32 or 64 bits).

Size of the objects of a floating point subtype

An object of a floating point subtype has the same size as its subtype.

Alignment of a floating point subtype

A floating point subtype is always word aligned.

Address of an object of a floating point subtype

Provided its alignment is not constrained by a record representation clause or a pragma PACK, the address of an object of a floating point subtype is a multiple of the alignment of the corresponding subtype.

4.4 Fixed Point Types

Small of a fixed point type

If no specification of small applies to a fixed point type, then the value of small is determined by the value of delta as defined by [3.5.9].

A specification of small can be used to impose a value of small. The value of small is required to be a power of two.

Predefined fixed point types

To implement fixed point types, the Alsys Compiler for the transputer uses a set of anonymous predefined types dependent upon the target transputer type.

For T4 and T8 transputer targets these anonymous types are:

```
type SHORT_FIXED is delta D range (-2**7-1)*S .. 2**7*S;
for SHORT_FIXED'SMALL use S;

type FIXED is delta D range (-2**31-1)*S .. 2**31*S;
for FIXED'SMALL use S;
```

For T2 transputer targets these anonymous types are:

```
type SHORT_FIXED is delta D range (-2**7-1)*S .. 2**7*S;
for SHORT_FIXED'SMALL use S;

type FIXED is delta D range (-2**15-1)*S .. 2**15*S;
for FIXED'SMALL use S;

type LONG_FIXED is delta D range (-2**31-1)*S .. 2**31*S;
for LONG_FIXED'SMALL use S;
```

where D is any real value and S any power of two less than or equal to D.

Selection of the parent of a fixed point type

A fixed point type declared by a declaration of the form:

```
type T is delta D range L .. R;
```

possibly with a small specification:

```
for T'SMALL use S;
```

is implicitly derived from a predefined fixed point type. The Compiler automatically selects the predefined fixed point type whose small and delta are the same as the small and delta of T and whose range is the shortest that includes the values L and R.

Encoding of fixed point values

In the program generated by the Compiler, a safe value V of a fixed point subtype F is represented as the integer:

V / F'BASE'SMALL

Minimum size of a fixed point subtype

The minimum size of a fixed point subtype is the minimum number of binary digits that is necessary for representing the values of the range of the subtype using the small of the base type (that is to say, in an unbiased form which includes a sign bit only if the range of the subtype includes negative values).

For a static subtype, if it has a null range its minimum size is 1. Otherwise, s and S being the bounds of the subtype, if i and I are the integer representations of m and M, the smallest and the greatest model numbers of the base type such that s < m and M < S, then the minimum size L is determined as follows. For i >= 0, L is the smallest positive integer such that I <= 2^L-1. For i < 0, L is the smallest positive integer such that -2^{L-1} <= i and I <= 2^{L-1}-1. For example:

```
type F is delta 2.0 range 0.0 .. 500.0;
-- The minimum size of F is 8 bits.

subtype S is F delta 16.0 range 0.0 .. 250.0;
-- The minimum size of S is 7 bits.

subtype D is S range X .. Y;
-- Assuming that X and Y are not static, the minimum size of D is 7 bits
-- (the same as the minimum size of its type mark S).
```

Size of a fixed point subtype

For T4 and T8 transputer targets, the sizes of the predefined fixed point types SHORT_FIXED and FIXED are 8 and 32 bits respectively. For T2 transputer targets, the sizes of the predefined fixed point types SHORT_FIXED, FIXED and LONG_FIXED are 8, 16 and 32 bits respectively.

When no size specification is applied to a fixed point type or to its first named subtype, its size and the size of any of its subtypes is the size of the predefined type from which it derives directly or indirectly. For example:

```
type F is delta 0.01 range 0.0 .. 1.0;
-- F is derived from a 8 bit predefined fixed type, its size is 8 bits.

type L is delta 0.01 range 0.0 .. 300.0;
-- L is derived from a 32 bit predefined fixed type, its size is 32 bits.

type N is new L range 0.0 .. 2.0;
-- N is indirectly derived from a 32 bit predefined fixed type, its size is 32 bits.
```

When a size specification is applied to a fixed point type, this fixed point type and each of its subtypes has the size specified by the length clause. The same rule applies to a first named subtype. The size specification must of course specify a value greater than or equal to the minimum size of the type or subtype to which it applies, for example:

```
type F is delta 0.01 range 0.0 .. 2.0;
for F'SIZE use 32;
-- F is derived from an 8 bit predefined fixed type, but its size is 32 bits
-- because of the size specification.

type L is delta 0.01 range 0.0 .. 300.0;
for F'SIZE use 16;
-- F is derived from a 32 bit predefined fixed type, but its size is 16 bits
-- because of the size specification.
-- The size specification is legal since the range contains no negative values
-- and therefore no sign bit is required.

type N is new F range 0.8 .. 1.0;
-- N is indirectly derived from a 16 bit predefined fixed type, but its size is
-- 32 bits because N inherits the size specification of F.
```

The Alsys Compiler implements size specifications. Nevertheless, as fixed point objects are represented using machine integers, the specified length cannot be greater than 32 bits.

Size of the objects of a fixed point subtype

Provided its size is not constrained by a record component clause or a pragma PACK, an object of a fixed point type has the same size as its subtype.

Alignment of a fixed point subtype

A fixed point subtype is byte aligned if its size is less than or equal to 8 bits, word aligned otherwise.

Address of an object of a fixed point subtype

Provided its alignment is not constrained by a record representation clause or a pragma PACK, the address of an object of a fixed point subtype is a multiple of the alignment of the corresponding subtype.

4.5 Access Types

Collection Size

When no specification of collection size applies to an access type, no storage space is reserved for its collection, and the value of the attribute STORAGE_SIZE is then 0.

As described in [13.2], a specification of collection size can be provided in order to reserve storage space for the collection of an access type. The Alsys Compiler fully implements this kind of specification.

Encoding of access values

Access values are machine addresses represented as machine word-sized values (i.e. 16 bits for T2 targets and 32 bits for T4 and T8 targets).

Minimum size of an access subtype

The minimum size of an access subtype is that of the word size of the target transputer.

Size of an access subtype

The size of an access subtype is the same as its minimum size.

The only size that can be specified for an access type using a size specification is its usual size.

Size of an object of an access subtype

An object of an access subtype has the same size as its subtype, thus an object of an access subtype is always one machine word long.

Alignment of an access subtype

An access subtype is always word aligned.

Address of an object of an access subtype

Provided its alignment is not constrained by a record representation clause or a pragma PACK, the address of an object of an access subtype is always on a word boundary. since its subtype is word aligned.

4.6 Task Types

Storage for a task activation

When no length clause is used to specify the storage space to be reserved for a task activation, the storage space indicated at bind time is used for this activation.

As described in [13.2], a length clause can be used to specify the storage space for the activation of each of the tasks of a given type. In this case the value indicated at bind time is ignored for this task type, and the length clause is obeyed.

It is not allowed to apply such a length clause to a derived type. The same storage space is reserved for the activation of a task of a derived type as for the activation of a task of the parent type.

Encoding of task values

Task values are represented as machine word sized values.

Minimum size of a task subtype

The minimum size of a task subtype is that of the word length of the target transputer.

Size of a task subtype

The size of a task subtype is the same as its minimum size.

The only size that can be specified for a task type using a size specification is its usual size.

Size of the objects of a task subtype

An object of a task subtype has the same size as its subtype. Thus an object of a task subtype is always one machine word long.

Alignment of a task subtype

A task subtype is always word aligned.

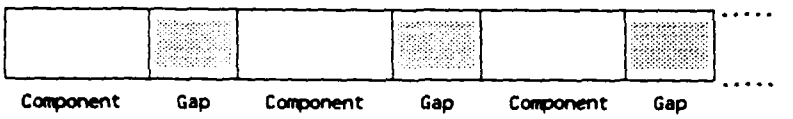
Address of an object of a task subtype

Provided its alignment is not constrained by a record representation clause, the address of an object of a task subtype is always on a word boundary, since its subtype is word aligned.

4.7 Array Types

Layout of an array

Each array is allocated in a contiguous area of storage units. All the components have the same size. A gap may exist between two consecutive components (and after the last one). All the gaps have the same size.



Components

If the array is not packed, the size of the components is the size of the subtype of the components, for example:

```
type A is array (1 .. 8) of BOOLEAN;
-- The size of the components of A is the size of the type BOOLEAN: 8 bits.

type DECIMAL_DIGIT is range 0 .. 9;
for DECIMAL_DIGIT'SIZE use 4;
type BINARY_CODED_DECIMAL is
      array (INTEGER range <>) of DECIMAL_DIGIT;
-- The size of the type DECIMAL_DIGIT is 4 bits. Thus in an array of
-- type BINARY_CODED_DECIMAL each component will be represented in
-- 4 bits as in the usual BCD representation.
```

If the array is packed and its components are neither records nor arrays, the size of the components is the minimum size of the subtype of the components, for example:

```
type A is array (1 .. 8) of BOOLEAN;
pragma PACK(A);
-- The size of the components of A is the minimum size of the type BOOLEAN:
-- 1 bit.
```

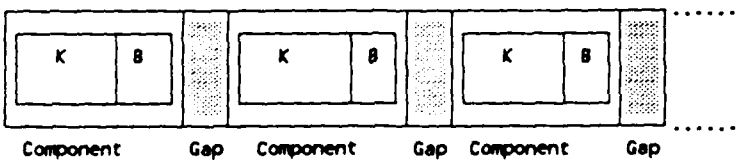
```
type DECIMAL_DIGIT is range 0 .. 9;
type BINARY_CODED_DECIMAL is
      array (INTEGER range <>) of DECIMAL_DIGIT;
pragma PACK(BINARY_CODED_DECIMAL);
-- The size of the type DECIMAL_DIGIT is 8 bits, but, as
-- BINARY_CODED_DECIMAL is packed, each component of an array of this
-- type will be represented in 4 bits as in the usual BCD representation.
```

Packing the array has no effect on the size of the components when the components are records or arrays.

Gaps

If the components are records or arrays, no size specification applies to the subtype of the components and the array is not packed, then the Compiler may choose a representation with a gap after each component; the aim of the insertion of such gaps is to optimize access to the array components and to their subcomponents. The size of the gap is chosen so that the relative displacement of consecutive components is a multiple of the alignment of the subtype of the components. This strategy allows each component and subcomponent to have an address consistent with the alignment of its subtype, for example:

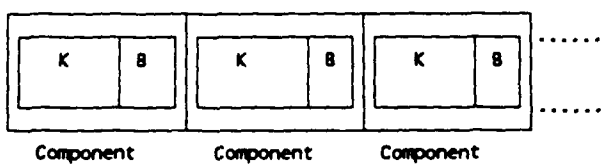
```
type INT is range -2**31 .. 2**31 - 1;
type R is
      record
            K : INT;          -- INT is word aligned.
            B : BOOLEAN;      -- BOOLEAN is byte aligned.
      end record;
-- Record type R is word aligned; its size is 40 bits.

type A is array (1 .. 10) of R;
-- A gap is inserted after each component in order to respect the
-- alignment of type R.
```



Array of type A: each subcomponent K has a word offset.

If a size specification applies to the subtype of the components or if the array is packed, no gaps are inserted, for example:

```
type INT is range -2**31 .. 2**31 - 1;
type R is
      record
            K : INT;
            B : BOOLEAN;
      end record;

type A is array (1 .. 10) of R;
pragma PACK(A);
-- There is no gap in an array of type A because A is packed.
-- The size of an object of type A will be 400 bits.

type NR is new R;
for NR'SIZE use 40;

type B is array (1 .. 10) of NR;
-- There is no gap in an array of type B because NR has a size specification.
-- The size of an object of type B will be 400 bits.
```



Array of type A or B: a subcomponent K can have any byte offset.

Size of an array subtype

The size of an array subtype is obtained by multiplying the number of its components by the sum of the size of the components and the size of the gaps (if any). If the subtype is unconstrained, the maximum number of components is considered.

The size of an array subtype cannot be computed at compile time

- if it has non-static constraints or is an unconstrained array type with non-static index subtypes (because the number of components can then only be determined at run time).
- if the components are records or arrays and their constraints or the constraints of their subcomponents (if any) are not static (because the size of the components and the size of the gaps can then only be determined at run time).

As has been indicated above, the effect of a pragma PACK on an array type is to suppress the gaps and to reduce the size of the components. The consequence of packing an array type is thus to reduce its size.

If the components of an array are records or arrays and their constraints or the constraints of their subcomponents (if any) are not static, the Compiler ignores any pragma PACK applied to the array type but issues a warning message. Apart from this limitation, array packing is fully implemented by the Alsys Compiler.

The only size that can be specified for an array type or first named subtype using a size specification is its usual size. Nevertheless, such a length clause can be useful to verify that the layout of an array is as expected by the application.

Size of the objects of an array subtype

The size of an object of an array subtype is always equal to the size of the subtype of the object.

Alignment of an array subtype

If no pragma PACK applies to an array subtype and no size specification applies to its components, the array subtype has the same alignment as the subtype of its components.

If a pragma PACK applies to an array subtype or if a size specification applies to its components (so that there are no gaps), the alignment of the array subtype is the lesser of the alignment of the subtype of its components and the relative displacement of the components.

Address of an object of an array subtype

Provided its alignment is not constrained by a record representation clause, the address of an object of an array subtype is a multiple of the alignment of the corresponding subtype.

4.8 Record Types

Layout of a record

Each record is allocated in a contiguous area of storage units. The size of a record component depends on its type. Gaps may exist between some components.

The positions and the sizes of the components of a record type object can be controlled using a record representation clause as described in [13.4]. In the Alsys implementation for transputer targets there is no restriction on the position that can be specified for a component of a record. If a component is not a record or an array, its size can be any size from the minimum size to the size of its subtype. If a component is a record or an array, its size must be the size of its subtype.

A record representation clause need not specify the position and the size for every component.

If no component clause applies to a component of a record, its size is the size of its subtype. Its position is chosen by the Compiler so as to optimize access to the components of the record: the offset of the component is chosen as a multiple of the alignment of the component subtype. Moreover, the Compiler chooses the position of the component so as to reduce the number of gaps and thus the size of the record objects.

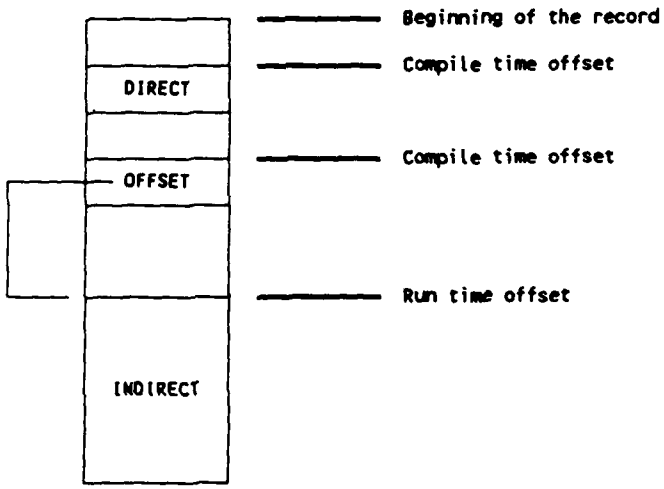
Because of these optimisations, there is no connection between the order of the components in a record type declaration and the positions chosen by the Compiler for the components in a record object.

Pragma PACK has no further effect on records. The Alsys Compiler always optimizes the layout of records as described above.

In the current version, it is not possible to apply a record representation clause to a derived type. The same storage representation is used for an object of a derived type as for an object of the parent type.

Indirect components

If the offset of a component cannot be computed at compile time, this offset is stored in the record objects at run time and used to access the component. Such a component is said to be indirect while other components are said to be direct:

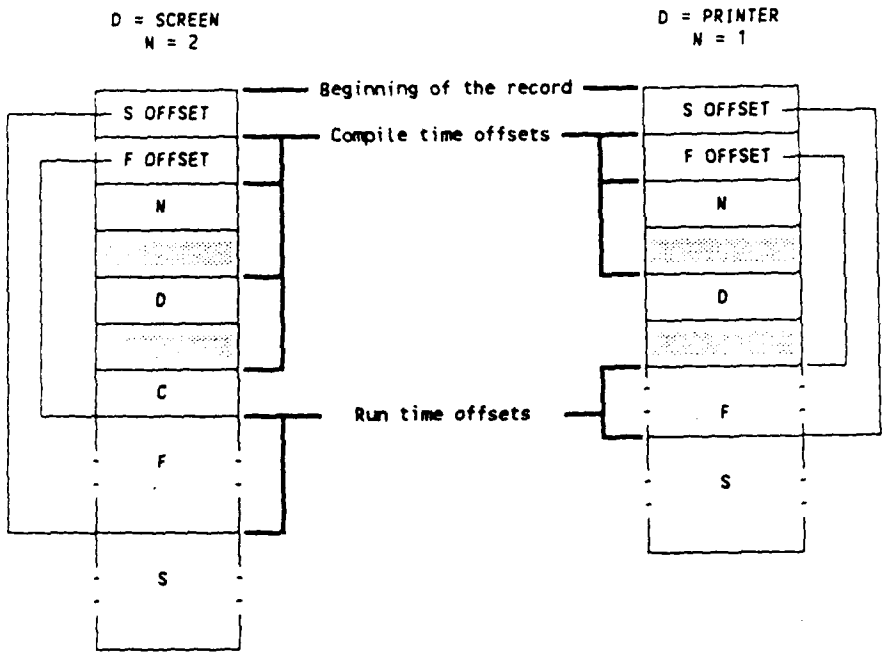


A direct and an indirect component

If a record component is a record or an array, the size of its subtype may be evaluated at run time and may even depend on the discriminants of the record. We will call these components dynamic components. For example:

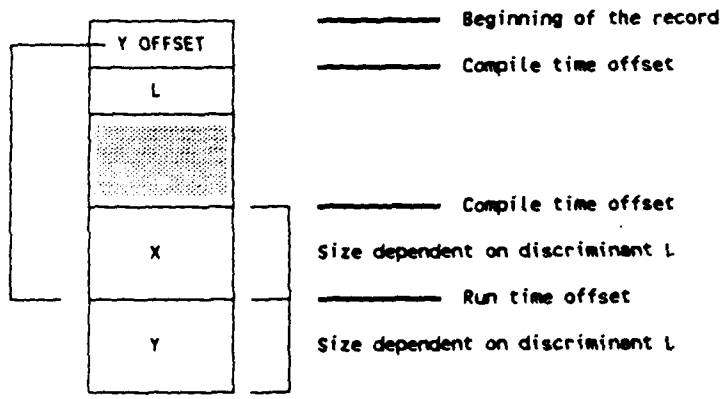
```
type DEVICE is (SCREEN, PRINTER);

type COLOUR is (GREEN, RED, BLUE);

type SERIES is array (POSITIVE range <>) of INTEGER;

type GRAPH (L : NATURAL) is
      record
            X : SERIES(1 .. L); -- The size of X depends on L
            Y : SERIES(1 .. L); -- The size of Y depends on L
      end record;

Q : POSITIVE;

type PICTURE (N : NATURAL; D : DEVICE) is
      record
            F : GRAPH(N); -- The size of F depends on N
            S : GRAPH(Q); -- The size of S depends on Q
            case D is
                  when SCREEN =>
                        C : COLOUR;
                  when PRINTER =>
                        null;
            end case;
      end record;
```

Any component placed after a dynamic component has an offset which cannot be evaluated at compile time and is thus indirect. In order to minimize the number of indirect components, the Compiler groups the dynamic components together and places them at the end of the record:



The record type PICTURE: F and S are placed at the end of the record

Thanks to this strategy, the only indirect components are dynamic components. But not all dynamic components are necessarily indirect: if there are dynamic components in a component list which is not followed by a variant part, then exactly one dynamic component of this list is a direct component because its offset can be computed at compilation time, for example :



The record type GRAPH: the dynamic component X is a direct component.

The offset of an indirect component is always expressed in storage units.

The space reserved for the offset of an indirect component must be large enough to store the size of any value of the record type (the maximum potential offset). The

Compiler evaluates an upper bound MS of this size and treats an offset as a component having an anonymous integer type whose range is 0 .. MS.

If C is the name of an indirect component, then the offset of this component can be denoted in a component clause by the implementation generated name C'OFFSET.

Implicit components

In some circumstances, access to an object of a record type or to its components involves computing information which only depends on the discriminant values. To avoid useless recomputation, the Compiler stores this information in the record objects, updates it when the values of the discriminants are modified and uses it when the objects or their components are accessed. This information is stored in special components called implicit components.

An implicit component may contain information which is used when the record object or several of its components are accessed. In this case the component will be included in any record object (the implicit component is considered to be declared before any variant part in the record type declaration). There can be two components of this kind; one is called RECORD_SIZE and the other VARIANT_INDEX.

On the other hand an implicit component may be used to access a given record component. In this case the implicit component exists whenever the record component exists (the implicit component is considered to be declared at the same place as the record component). Components of this kind are called ARRAY_DESCRIPTORs or RECORD_DESCRIPTORs.

- *RECORD_SIZE*

This implicit component is created by the Compiler when the record type has a variant part and its discriminants are defaulted. It contains the size of the storage space necessary to store the current value of the record object (note that the storage effectively allocated for the record object may be more than this).

The value of a RECORD_SIZE component may denote a number of bits or a number of storage units. In general it denotes a number of storage units, but if any component clause specifies that a component of the record type has an offset or a size which cannot be expressed using storage units, then the value designates a number of bits.

The implicit component RECORD_SIZE must be large enough to store the maximum size of any value of the record type. The Compiler evaluates an upper bound MS of this size and then considers the implicit component as having an anonymous integer type whose range is 0 .. MS.

If R is the name of the record type, this implicit component can be denoted in a component clause by the implementation generated name R'RECORD_SIZE.

- *VARIANT_INDEX*

This implicit component is created by the Compiler when the record type has a variant part. It indicates the set of components that are present in a record value. It is used when a discriminant check is to be done.

Component lists that do not contain a variant part are numbered. These numbers are the possible values of the implicit component VARIANT_INDEX. For example:

```
type VEHICLE is (AIRCRAFT, ROCKET, BOAT, CAR);

type DESCRIPTION (KIND : VEHICLE := CAR) is
      record
            SPEED : INTEGER;
            case KIND is
                  when AIRCRAFT | CAR =>
                        WHEELS : INTEGER;
                        case KIND is
                              when AIRCRAFT =>     -- 1
                                    WINGSPAN : INTEGER;
                              when others =>       -- 2
                                    null;
                        end case;
                  when BOAT =>                     -- 3
                        STEAM  : BOOLEAN;
                  when ROCKET =>                   -- 4
                        STAGES : INTEGER;
            end case;
      end record;
```

The value of the variant index indicates the set of components that are present in a record value:

Variant Index	Set
1	(KIND, SPEED, WHEELS, WINGSPAN)
2	(KIND, SPEED, WHEELS)
3	(KIND, SPEED, STEAM)
4	(KIND, SPEED, STAGES)

A comparison between the variant index of a record value and the bounds of an interval is enough to check that a given component is present in the value:

Component	Interval
KIND	--
SPEED	--
WHEELS	1 .. 2
WINGSPAN	1 .. 1
STEAM	3 .. 3
STAGES	4 .. 4

The implicit component VARIANT_INDEX must be large enough to store the number V of component lists that don't contain variant parts. The Compiler treats this implicit component as having an anonymous integer type whose range is 1 .. V.

If R is the name of the record type, this implicit component can be denoted in a component clause by the implementation generated name R'VARIANT_INDEX.

- *ARRAY_DESCRIPTOR*

An implicit component of this kind is associated by the Compiler with each record component whose subtype is an anonymous array subtype that depends on a discriminant of the record. It contains information about the component subtype.

The structure of an implicit component of kind ARRAY_DESCRIPTOR is not described in this documentation. Nevertheless, if a programmer is interested in specifying the location of a component of this kind using a component clause, he can obtain the size of the component using the ASSEMBLY parameter in the COMPILE command.

The Compiler treats an implicit component of the kind ARRAY_DESCRIPTOR as having an anonymous record type. If C is the name of the record component whose subtype is described by the array descriptor, then this implicit component can be denoted in a component clause by the implementation generated name C'ARRAY_DESCRIPTOR.

- *RECORD_DESCRIPTOR*

An implicit component of this kind is associated by the Compiler with each record component whose subtype is an anonymous record subtype that depends on a discriminant of the record. It contains information about the component subtype.

The structure of an implicit component of kind RECORD_DESCRIPTOR is not described in this documentation. Nevertheless, if a programmer is interested in specifying the location of a component of this kind using a component clause, he can obtain the size of the component using the ASSEMBLY parameter in the COMPILE command.

The Compiler treats an implicit component of the kind RECORD_DESCRIPTOR as having an anonymous record type. If C is the name of the record component whose subtype is described by the record descriptor, then this implicit component can be denoted in a component clause by the implementation generated name C'RECORD_DESCRIPTOR.

Suppression of implicit components

The Alsys implementation provides the capability of suppressing the implicit components RECORD_SIZE and/or VARIANT_INDEX from a record type. This can be done using an implementation defined pragma called IMPROVE. The syntax of this pragma is as follows:

pragma IMPROVE (TIME | SPACE , [ON =>] *simple_name*);

The first argument specifies whether TIME or SPACE is the primary criterion for the choice of the representation of the record type that is denoted by the second argument.

If TIME is specified, the Compiler inserts implicit components as described above. If on the other hand SPACE is specified, the Compiler only inserts a VARIANT_INDEX or a RECORD_SIZE component if this component appears in a record representation

clause that applies to the record type. A record representation clause can thus be used to keep one implicit component while suppressing the other.

A pragma IMPROVE that applies to a given record type can occur anywhere that a representation clause is allowed for this type.

Size of a record subtype

Unless a component clause specifies that a component of a record type has an offset or a size which cannot be expressed using storage units, the size of a record subtype is rounded up to a whole number of storage units.

The size of a constrained record subtype is obtained by adding the sizes of its components and the sizes of its gaps (if any). This size is not computed at compile time

- when the record subtype has non-static constraints,
- when a component is an array or a record and its size is not computed at compile time.

The size of an unconstrained record subtype is obtained by adding the sizes of the components and the sizes of the gaps (if any) of its largest variant. If the size of a component or of a gap cannot be evaluated exactly at compile time, an upper bound of this size is used by the Compiler to compute the subtype size.

The only size that can be specified for a record type or first named subtype using a size specification is its usual size. Nevertheless, such a length clause can be useful to verify that the layout of a record is as expected by the application.

Size of an object of a record subtype

An object of a constrained record subtype has the same size as its subtype.

An object of an unconstrained record subtype has the same size as its subtype if this size is less than or equal to 8 Kbyte. If the size of the subtype is greater than this, the object has the size necessary to store its current value; storage space is allocated and released as the discriminants of the record change.

Alignment of a record subtype

When no record representation clause applies to its base type, a record subtype has the same alignment as the component with the highest alignment requirement.

When a record representation clause that does not contain an alignment clause applies to its base type, a record subtype has the same alignment as the component with the highest alignment requirement which has not been overridden by its component clause.

When a record representation clause that contains an alignment clause applies to its base type, a record subtype has an alignment that obeys the alignment clause.

Address of an object of a record subtype

Provided its alignment is not constrained by a representation clause, the address of an object of a record subtype is a multiple of the alignment of the corresponding subtype.

5 Conventions for Implementation-Generated Names

Special record components are introduced by the Compiler for certain record type definitions. Such record components are implementation-dependent; they are used by the Compiler to improve the quality of the generated code for certain operations on the record types. The existence of these components is established by the Compiler depending on implementation-dependent criteria. Attributes are defined for referring to them in record representation clauses. An error message is issued by the Compiler if the user refers to an implementation-dependent component that does not exist. If the implementation-dependent component exists, the Compiler checks that the storage location specified in the component clause is compatible with the treatment of this component and the storage locations of other components. An error message is issued if this check fails.

There are four such attributes:

T'RECORD_SIZE
: For a prefix T that denotes a record type. This attribute refers to the record component introduced by the Compiler in a record to store the size of the record object. This component exists for objects of a record type with defaulted discriminants when the sizes of the record objects depend on the values of the discriminants.

T'VARIANT_INDEX
: For a prefix T that denotes a record type. This attribute refers to the record component introduced by the Compiler in a record to assist in the efficient implementation of discriminant checks. This component exists for objects of a record type with variant type.

C'ARRAY_DESCRIPTOR
: For a prefix C that denotes a record component of an array type whose component subtype definition depends on discriminants. This attribute refers to the record component introduced by the Compiler in a record to store information on subtypes of components that depend on discriminants.

C'RECORD_DESCRIPTOR
: For a prefix C that denotes a record component of a record type whose component subtype definition depends on discriminants. This attribute refers to the record component introduced by the Compiler in a record to store information on subtypes of components that depend on discriminants.

6 Address Clauses

6.1 Address Clauses for Objects

An address clause can be used to specify an address for an object as described in [13.5]. When such a clause applies to an object no storage is allocated for it in the program generated by the Compiler. The program accesses the object using the address specified in the clause.

An address clause is not allowed for task objects, nor for unconstrained records whose size is greater than 8 Kbyte.

6.2 Address Clauses for Program Units

Address clauses for program units are not implemented in the current version of the Compiler.

6.3 Address Clauses for Entries

Address clauses for entries are not implemented in the current version of the Compiler.

7 Restrictions on Unchecked Conversions

Unconstrained arrays are not allowed as target types.

Unconstrained record types without defaulted discriminants are not allowed as target types.

If the source and the target types are each scalar or access types, the sizes of the objects of the source and target types must be equal. If a composite type is used either as the source type or as the target type this restriction on the size does not apply.

If the source and the target types are each of scalar or access type or if they are both of composite type, the effect of the function is to return the operand.

In other cases the effect of unchecked conversion can be considered as a copy:

- if an unchecked conversion is achieved of a scalar or access source type to a composite target type, the result of the function is a copy of the source operand: the result has the size of the source.

- if an unchecked conversion is achieved of a composite source type to a scalar or access target type, the result of the function is a copy of the source operand: the result has the size of the target.

8 Input-Output Packages

The predefined input-output packages SEQUENTIAL_IO [14.2.3], DIRECT_IO [14.2.5], and TEXT_IO [14.3.10] are implemented as described in the Language Reference Manual, as is the package IO_EXCEPTIONS [14.5], which specifies the exceptions that can be raised by the predefined input-output packages.

The package LOW_LEVEL_IO [14.6], which is concerned with low-level machine-dependent input-output, has not been implemented.

All accesses to the services of the host system are provided through the INMOS supplied *iserver* tool, so much of Ada input-output is host independent.

8.1 NAME Parameter

No special treatment is applied to the NAME parameter supplied to the Ada procedures CREATE or OPEN [14.2.1]. This parameter is passed immediately on to the INMOS server and from there to the host operating system. The file name can thus be in any format acceptable to the host system.

8.2 FORM Parameter

The FORM parameter comprises a set of attributes formulated according to the lexical rules of [2], separated by commas. The FORM parameter may be given as a null string except when DIRECT_IO is instantiated with an unconstrained type; in this case the RECORD_SIZE attribute must be provided. Attributes are comma-separated; blanks may be inserted between lexical elements as desired. In the descriptions below the meanings of *natural*, *positive*, etc., are as in Ada; attribute keywords (represented in upper case) are identifiers [2.3] and as such may be specified without regard to case.

USE_ERROR is raised if the FORM parameter does not conform to these rules.

The attributes are as follows:

File sharing attribute

This attribute allows control over the sharing of one external file between several internal files within a single program. In effect it establishes rules for subsequent OPEN and CREATE calls which specify the same external file. If such rules are violated or if a different file sharing attribute is specified in a later OPEN or CREATE call, USE_ERROR will be raised. The syntax is as follows:

NOT_SHARED |
SHARED => *access_mode*

where

access_mode ::= READERS | SINGLE_WRITER | ANY

A file sharing attribute of:

NOT_SHARED

implies only one internal file may access the external file.

SHARED => READERS

imposes no restrictions on internal files of mode IN_FILE, but prevents any internal files of mode OUT_FILE or INOUT_FILE being associated with the external file.

SHARED => SINGLE_WRITER

is as SHARED => READERS, but in addition allows a single internal file of mode OUT_FILE or INOUT_FILE.

SHARED => ANY

places no restrictions on external file sharing.

If a file of the same name has previously been opened or created, the default is taken from that file's sharing attribute, otherwise the default depends on the mode of the file: for mode IN_FILE the default is SHARED => READERS, for modes INOUT_FILE and OUT_FILE the default is NOT_SHARED.

Record size and record unit attributes

These attributes control the structure of external binary files.

A binary file can be viewed as a sequence (sequential access) or a set (direct access) of consecutive records, each of the following structure:

[HEADER] OBJECT [UNUSED_PART]

where:

- OBJECT is the exact binary representation of the Ada object in the executable program (possibly including an implicit object descriptor).
- HEADER contains two word sized values, the length of the object and the length of the descriptor.
- UNUSED_PART is a gap of variable size to permit full control of the record's size.

The HEADER is only implemented if the actual parameter of the instantiation of the IO package is unconstrained.

The formats of the file structure attributes are as follows:

RECORD_SIZE => size_in_bytes

RECORD_UNIT => size_in_bytes

In the case of DIRECT_IO for unconstrained types the user is required to specify the RECORD_SIZE attribute. However, for SEQUENTIAL_IO for unconstrained types the attribute is illegal. USE_ERROR will be raised by the OPEN or CREATE procedures if either of these checks fail.

In all cases the value given must not be smaller than a mininum size. For constrained types, this minimum size is ELEMENT_TYPE'SIZE / SYSTEM.STORAGE_UNIT; USE_ERROR will be raised if this rule is violated. For unconstrained types, the minimum size is ELEMENT_TYPE'DESCRIPTOR_SIZE / SYSTEM.STORAGE_UNIT plus the size of the largest record which is to be read or written. If a larger record is processed DATA_ERROR will be raised by the READ or WRITE.

If no RECORD_SIZE attribute is specified for constrained types, the default value of the object's size is assumed. In this case no UNUSED_PART will be implemented.

The RECORD_UNIT attribute is only applicable to SEQUENTIAL_IO for unconstrained types; it has a default value of 1. If specified, the record size will be the smallest multiple of this value that holds the object and its length. This is the only case where a file may contain variable length records.

Buffer size attribute

This attribute controls the size of the buffer used as a cache for input-output operations:

BUFFER_SIZE => size_in_bytes

The default value for BUFFER_SIZE is 0, which means no buffering.

Append

This attribute may only be used in the FORM parameter of the OPEN command. If used in the FORM parameter of the CREATE command, USE_ERROR will be raised.

The affect of this attribute is to cause writing to commence at the end of the existing file.

The syntax of the APPEND attribute is simply:

APPEND

The default is APPEND => FALSE, but this is over-ridden if this attribute is specified.

8.3 USE_ERROR

The following conditions will cause USE_ERROR to be raised:

- Specifying a FORM parameter whose syntax does not conform to the rules given above.

- Specifying the RECORD_SIZE FORM parameter attribute to have a value of zero, or failing to specify RECORD_SIZE for instantiations of DIRECT_IO for unconstrained types.

- Specifying a RECORD_SIZE FORM parameter attribute to have a value less than that required to hold the element for instantiations of DIRECT_IO and SEQUENTIAL_IO for constrained types.

- Violating the file sharing rules stated above.

- Attempting to perform an input-output operation which is not supported by the INMOS iserver due to restrictions of the host operating system.

- Errors detected whilst reading or writing (e.g. writing to a file on a read-only disk).

9 Characteristics of Numeric Types

9.1 Integer Types - T2 transputer targets

The ranges of values for integer types for T2 transputer targets declared in package STANDARD are as follows:

```
SHORT_INTEGER       -128 .. 127                         -- 2**7 - 1
INTEGER             -32768 .. 32767                     -- 2**15 - 1
LONG_INTEGER        -2147483648 .. 2147483647           -- 2**31 - 1
```

9.2 Integer Types - T4/T8 transputer targets

The ranges of values for integer types for T4 and T8 transputer targets declared in package STANDARD are as follows:

```
SHORT_INTEGER       -128 .. 127                         -- 2**7 - 1
INTEGER             -2147483648 .. 2147483647           -- 2**31 - 1
```

9.3 Other Integer Types

For the packages DIRECT_IO and TEXT_IO, the ranges of values for types COUNT and POSITIVE_COUNT are as follows:

```
COUNT               0 .. 2147483647                     -- 2**31 - 1
POSITIVE_COUNT      1 .. 2147483647                     -- 2**31 - 1
```

For the package TEXT_IO, the range of values for the type FIELD is as follows:

```
FIELD               0 .. 255                            -- 2**8 - 1
```

9.4 Floating Point Type Attributes

FLOAT

		Approximate value
DIGITS	6	
MANTISSA	21	
EMAX	84	
EPSILON	2.0 ** -20	9.54E-7
SMALL	2.0 ** -85	2.58E-26
LARGE	2.0 ** 84 * (1.0 - 2.0 ** -21)	1.93E+25
SAFE_EMAX	125	
SAFE_SMALL	2.0 ** -126	1.18E-38
SAFE_LARGE	2.0 ** 125 * (1.0 - 2.0 ** -21)	4.25E+37
FIRST	-2.0 ** 127 * (2.0 - 2.0 ** -23)	-3.40E+38
LAST	2.0 ** 127 * (2.0 - 2.0 ** -23)	3.40E+38
MACHINE_RADIX	2	
MACHINE_MANTISSA	24	
MACHINE_EMAX	128	
MACHINE_EMIN	-125	
MACHINE_ROUNDS	TRUE	
MACHINE_OVERFLOWS	TRUE	
SIZE	32	

LONG_FLOAT

		Approximate value
DIGITS	15	
MANTISSA	51	
EMAX	204	
EPSILON	2.0 ** -50	8.88E-16
SMALL	2.0 ** -205	1.94E-62
LARGE	2.0 ** 204 * (1.0 - 2.0 ** -51)	2.57E+61
SAFE_EMAX	1021	
SAFE_SMALL	2.0 ** -1022	2.22E-308
SAFE_LARGE	2.0 ** 1021 * (1.0 - 2.0 ** -51)	2.25E+307
FIRST	-2.0 ** 1023 * (2.0 - 2.0 ** -51)	-1.79E+308
LAST	2.0 ** 1023 * (2.0 - 2.0 ** -51)	1.79E+308
MACHINE_RADIX	2	
MACHINE_MANTISSA	53	
MACHINE_EMAX	1024	
MACHINE_EMIN	-1021	
MACHINE_ROUNDS	TRUE	
MACHINE_OVERFLOWS	TRUE	
SIZE	64	

9.5 *Attributes of Type DURATION*

DURATION'DELTA	2.0 ** -14
DURATION'SMALL	2.0 ** -14
DURATION'LARGE	131072.0
DURATION'FIRST	-86400.0
DURATION'LAST	86400.0

INDEX

ADDRESS attribute 7
  restrictions 7
Append attribute 38
ARRAY_DESCRIPTOR attribute 33
ASCII 4, 5
Attributes 7
  ARRAY_DESCRIPTOR 33
  DESCRIPTOR_SIZE 7
  IS_ARRAY 7
  RECORD_DESCRIPTOR 33
  RECORD_SIZE 33, 36
  representation attributes 7
  VARIANT_INDEX 33

BOOLEAN 4
Buffer_size attribute 38

CHARACTER 4
COUNT 40

DESCRIPTOR_SIZE attribute 7, 38
DIRECT_IO 36, 40
DURATION
  attributes 42

Enumeration types 4
  BOOLEAN 4
  CHARACTER 4

FIELD 40
File sharing attribute 36
Fixed point types 4
  DURATION 42
FLOAT 4, 41
Floating point types 4
  FLOAT 4, 41
  LONG_FLOAT 4, 41
FORM parameter 36
FORM parameter attributes
  append 38
  buffer_size attribute 38
  file sharing attribute 36
  record_size attribute 37, 39
  record_unit attribute 37

Implementation-dependent attributes 7
Implementation-dependent pragma 2
Implementation-generated names 33
IMPROVE 6
INLINE 2
Input-Output packages 36
  DIRECT_IO 36
  IO_EXCEPTIONS 36
  LOW_LEVEL_IO 36
  SEQUENTIAL_IO 36
  TEXT_IO 36
INTEGER 4, 40
Integer types 4, 40
  COUNT 40
  FIELD 40
  INTEGER 4, 40
  LONG_INTEGER 4, 40
  POSITIVE_COUNT 40
  SHORT_INTEGER 4, 40
INTERFACE 2
INTERFACE_NAME 2, 6
Interfaced subprograms
  Restrictions 5
IO_EXCEPTIONS 36
IS_ARRAY attribute 7

Language_name 2
LONG_FLOAT 4, 41
LONG_INTEGER 4, 40
LOW_LEVEL_IO 36

NAME parameter 36
NOT_SHARED 36
Numeric types
  characteristics 40
  Fixed point types 42
  integer types 40

OCCAM 2

PACK 6
Parameter representations 3
  Access types 5
  Array types 5
  Enumeration types 4
  Fixed point types 4
  Floating point types 4
  Integer types 4
  Record types 5
Parameter-passing conventions 3
POSITIVE_COUNT 40
Pragma INLINE 2
Pragma INTERFACE 2
  language_name 2
  OCCAM 2
  subprogram_name 2
Pragma INTERFACE_NAME 2
  string_literal 6
  subprogram_name 6
Pragmas
  IMPROVE 6

INTERFACE 2
INTERFACE_NAME 6
PACK 6
PRIORITY 6
SUPPRESS 6
PRIORITY 6

RECORD_DESCRIPTOR attribute 33
RECORD_SIZE attribute 33, 36, 37, 39
Record_unit attribute 37
Representation attributes 7
Representation clauses 9
restrictions 9

SEQUENTIAL_IO 36
SHARED 36
SHORT_INTEGER 4, 40
STRING 5
String literal 6
Subprogram_name 2, 6
SUPPRESS 6
SYSTEM package 8

TEXT_IO 36, 40

Unchecked conversions 35
restrictions 35
USE_ERROR 36, 38

VARIANT_INDEX attribute 33

APPENDIX C

TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below:

Name and Meaning	Value
$ACC_SIZE An integer literal whose value is the number of bits sufficient to hold any value of an access type.	32
$BIG_ID1 Identifier the size of the maximum input line length with varying last character.	(1..254=>'A', 255=>1)
$BIG_ID2 Identifier the size of the maximum input line length with varying last character.	(1..254=>'A', 255=>2)
$BIG_ID3 Identifier the size of the maximum input line length with varying middle character.	(1..127=>'A', 128=>3, 129..255=>'A')
$BIG_ID4 Identifier the size of the maximum input line length with varying middle character.	(1..127=>'A', 128=>4, 129..255=>'A')
$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length	(1..252=>0, 253..255=>298)

$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	(1..249=>0, 250..255=>69.0E1)
$BIG_STRING1 A string literal which when catenated with BIG_STRING2 yields the image of BIG_ID1.	(1..127=>'A')
$BIG_STRING2 A string literal which when catenated to the end of BIG_STRING1 yields the image of BIG_ID1.	(1..127=>'A', 128=>1)
$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	(1..235=>' ')
$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2147483647
$DEFAULT_MEM_SIZE An integer literal whose value is SYSTEM.MEMORY_SIZE.	2147483647
$DEFAULT_STOR_UNIT An integer literal whose value is SYSTEM.STORAGE_UNIT.	8
$DEFAULT_SYS_NAME The value of the constant SYSTEM.SYSTEM_NAME.	TRANSPUTER
$DELTA_DOC A real literal whose value is SYSTEM.FINE_DELTA.	2#1.0#E-31
$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	255

$FIXED_NAME The name of a predefined fixed-point type other than DURATION.	NO_SUCH_TYPE
$FLOAT_NAME The name of a predefined floating-point type other than FLOAT, SHORT_FLOAT, or LONG_FLOAT.	NO_SUCH_TYPE
$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	100000.0
$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	10000000.0
$HIGH_PRIORITY An integer literal whose value is the upper bound of the range for the subtype SYSTEM.PRIORITY.	10
$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	?#~@[()]+=
$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	[()]+=?3~@'
$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-2147483648
$INTEGER_LAST A universal integer literal whose value is INTEGER'LAST.	2147483647
$INTEGER_LAST_PLUS_1 A universal integer literal whose value is INTEGER'LAST+1.	2147483648

$LESS_THAN_DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-100000.0
$LESS_THAN_DURATION_BASE_FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-10000000.0
$LOW_PRIORITY An integer literal whose value is the lower bound of the range for the subtype SYSTEM.PRIORITY.	1
$MANTISSA_DOC An integer literal whose value is SYSTEM.MAX_MANTISSA.	31
$MAX_DIGITS Maximum digits supported for floating-point types.	15
$MAX_IN_LEN Maximum input line length permitted by the implementation.	255
$MAX_INT A universal integer literal whose value is SYSTEM.MAX_INT.	2147483647
$MAX_INT_PLUS_1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	2147483648
$MAX_LEN_INT_BASED_LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	(1..2=>'2:', 3..252=>'0', 253..255=>'11:')
$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be MAX_IN_LEN long.	(1..3=>'16:', 4..251=>'0', 252..255=>'F.E:')

$MAX_STRING_LITERAL A string literal of size MAX_IN_LEN, including the quote characters.	(1=>'"', 2..254=>'A', 255=>'"')
$MIN_INT A universal integer literal whose value is SYSTEM.MIN_INT.	-2147483648
$MIN_TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has no entries, no declarations, and "NULL;" as the only statement in its body.	32
$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	NO_SUCH_TYPE
$NAME_LIST A list of enumeration literals in the type SYSTEM.NAME, separated by commas.	TRANSPUTER
$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#FFFFFFFF#
$NEW_MEM_SIZE An integer literal whose value is a permitted argument for pragma memory_size, other than $DEFAULT_MEM_SIZE. If there is no other value, then use $DEFAULT_MEM_SIZE.	2147483647

$NEW_STOR_UNIT An integer literal whose value is a permitted argument for pragma storage_unit, other than $DEFAULT_STOR_UNIT. If there is no other permitted value, then use value of SYSTEM.STORAGE_UNIT.	8
$NEW_SYS_NAME A value of the type SYSTEM.NAME, other than $DEFAULT_SYS_NAME. If there is only one value of that type, then use that value.	TRANSPUTER
$TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has a single entry with one inout parameter.	32
$TICK A real literal whose value is SYSTEM.TICK.	1.0E-6

APPENDIX D

WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following **44** tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form AI-ddddd is to an Ada Commentary.

E28005C This test expects that the string "-- TOP OF PAGE. --63" of line 204 will appear at the top of the listing page due to a pragma PAGE in line 203; but line 203 contains text that follows the pragma, and it is this that must appear at the top of the page.

A39005G This test unreasonably expects a component clause to pack an array component into a minimum size (line 30).

B97102E This test contains an unintended illegality: a select statement contains a null statement at the place of a selective wait alternative (line 31).

C97116A This test contains race conditions, and it assumes that guards are evaluated indivisibly. A conforming implementation may use interleaved execution in such a way that the evaluation of the guards at lines 50 & 54 and the execution of task CHANGING_OF_THE_GUARD results in a call to REPORT.FAILED at one of lines 52 or 56.

BC3009B This test wrongly expects that circular instantiations will be detected in several compilation units even though none of the units is illegal with respect to the units it depends on; by AI-00256, the illegality need not be detected until execution is attempted (line 95).

CD2A62D This test wrongly requires that an array object's size be no greater than 10 although its subtype's size was specified to be 40 (line 137).

CD2A63A..D, CD2A66A..D, CD2A73A..D, CD2A76A..D [16 tests]
These tests wrongly attempt to check the size of objects of a derived type (for which a 'SIZE length clause is given) by passing them to a derived subprogram (which implicitly converts them to the parent type (Ada standard 3.4:14)). Additionally, they use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

CD2A81G, CD2A83G, CD2A84N & M, & CD5011O [5 tests]
These tests assume that dependent tasks will terminate while the main program executes a loop that simply tests for task termination; this is not the case, and the main program may loop indefinitely (lines 74, 85, 86 & 96, 86 & 96, and 58, resp.).

CD2B15C & CD7205C

These tests expect that a 'STORAGE_SIZE length clause provides precise control over the number of designated objects in a collection; the Ada standard 13.2:15 allows that such control must not be expected.

CD2D11B This test gives a SMALL representation clause for a derived fixed-point type (at line 30) that defines a set of model numbers that are not necessarily represented in the parent type; by Commentary AI-00099, all model numbers of a derived fixed-point type must be representable values of the parent type.

CD5007B This test wrongly expects an implicitly declared subprogram to be at the the address that is specified for an unrelated subprogram (line 303).

ED7004B, ED7005C & D, ED7006C & D [5 tests]

These tests check various aspects of the use of the three SYSTEM pragmas; the AVO withdraws these tests as being inappropriate for validation.

CD7105A This test requires that successive calls to CALENDAR.CLOCK change by at least SYSTEM.TICK; however, by Commentary AI-00201, it is only the expected frequency of change that must be at least SYSTEM.TICK--particular instances of change may be less (line 29).

CD7203B, & CD7204B

These tests use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.

CD7205D This test checks an invalid test objective: it treats the specification of storage to be reserved for a task's activation as though it were like the specification of storage for a collection.

CE2107I This test requires that objects of two similar scalar types be distinguished when read from a file--DATA_ERROR is expected to be raised by an attempt to read one object as of the other type. However, it is not clear exactly how the Ada standard 14.2.4:4 is to be interpreted; thus, this test objective is not considered valid. (line 90)

CE3111C This test requires certain behavior, when two files are associated with the same external file, that is not required by the Ada standard.

CE3301A This test contains several calls to END_OF_LINE & END_OF_PAGE that have no parameter: these calls were intended to specify a file, not to refer to STANDARD_INPUT (lines 103, 107, 118, 132, & 136).

CE3411B This test requires that a text file's column number be set to COUNT'LAST in order to check that LAYOUT_ERROR is raised by a subsequent PUT operation. But the former operation will generally raise an exception due to a lack of available disk space, and the test would thus encumber validation testing.